Assignment-Based Partitioning in a Condition Monitoring System

Yongqiang Huang and Hector Garcia-Molina

Stanford University, Stanford, CA 94305
{yhuang, hector}@cs.stanford.edu

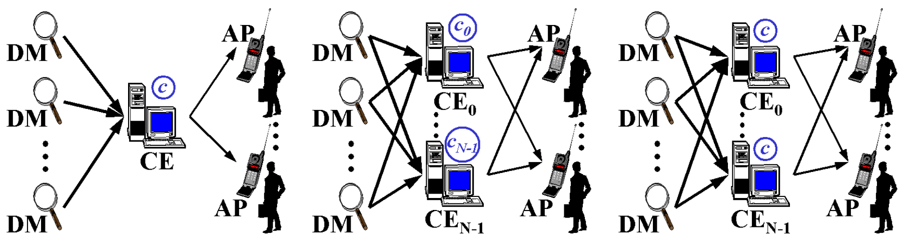
Abstract. A condition monitoring system tracks real-world variables and alerts users when a predefined condition becomes true, e.g., when enemy planes take off, or when suspicious terrorist activities and communication are detected. However, a monitoring server can easily get overwhelmed by rapid incoming data. To prevent this, we partition the condition being monitored and distribute the workload onto several independent servers. In this paper, we study the problem of how to make a partitioned system behave "equivalently" to a one-server system. We identify and formally define three desirable properties of a partitioned system, namely, orderedness, consistency, and completeness. We propose assignment-based partitioning as a solution that can handle opaque conditions and simplifies load balancing. We also look at a few typical partitioned systems, and discuss their merits using several metrics that we define. Finally, an algorithm is presented to reduce complex system configurations to simpler ones.

1 Introduction

A *condition monitoring system* is used to track the state of certain real-world variables and alert the users when pre-defined conditions about the variables are satisfied. For example, soldiers in a battlefield need to be notified when the location of enemy troops is within a certain range. Authorities must be alerted if suspicious money transfer transactions or communication messages are detected that fit into a "terrorist" pattern. The manager of a nuclear plant has to get a message on his/her Personal Data Assistant (PDA) whenever the temperature of the reactor is higher than a safety limit.

Figure 1(a) illustrates such a condition monitoring system. It consists of one or more *Data Monitors* (DM), a *Condition Evaluator* (CE), and one or more *Alert Presenters* (AP). A Data Monitor tracks the state of a real world variable, such as the reactor temperature. Periodically or whenever the variable changes, the DM sends out an *update*, i.e., a temperature reading. The stream of updates arrive at the Condition Evaluator, which uses them to evaluate a predefined user condition c, e.g., "reactor temperature is over 3000 degrees." If the condition c is satisfied, an *alert* is sent to the Alert Presenter, which is responsible for alerting the user. In this case, the user will be notified by a message on his/her PDA that the reactor has overheated. If the PDA is off or disconnected, the CE logs the alert, and sends it later, when the AP becomes available.

We call systems with a single Condition Evaluator, such as the one in Figure 1(a), *centralized* monitoring systems. One problem with a centralized system is that the CE can easily get overloaded [1]. High volume of updates may arrive at the CE at a rapid pace. For each new update received, the CE needs to match it against c. The matching can become a time-consuming operation if, for example, the CE needs to extract enemy



(a) Centralized (b) Partitioned (c) Replicated

Fig. 1: Condition monitoring systems

plane movement information from satellite images, or to analyze text messages in order to determine the topic. Furthermore, for each update that matches, an alert has to be generated, logged, and sent out to all interested APs. In a system with frequent new updates and many users, the resources required to match updates and to send alerts may well exceed the capability of a single machine.

The above problem can be alleviated by introducing multiple independent Condition Evaluators to share the workload (Figure 1(b)). In a *partitioned* monitoring system, each CE$_i$ monitors a modified "condition partition" c_i instead of c. The CEs independently make their own decisions about when to generate an alert. Ideally, the work at each CE$_i$ is reduced to a fraction of the centralized case, while the outcome is kept "equivalent."

One way of obtaining the c_i's is to break up c into sub-expressions. For example, assume c is "temperature is over 3000 degrees OR temperature has risen for more than 200 degrees since last reading." We can then split the disjunction to obtain c_0 ("temperature is over 3000 degrees") and c_1 ("temperature has risen for more than 200 degrees"). Such *structure-based partitioning* takes advantage of knowledge about c's internal structure, and the resulting condition partitions are usually subexpressions of c.

Structure-based partitioning often gives a natural and efficient way of breaking up c. However, this type of partitioning is only applicable when the internal structure of c is known and amenable to being broken apart. Moreover, it is often difficult to evenly balance the workload among the CEs because one condition partition may be much more costly to monitor than another. Instead, we propose a different approach, suitable for partitioning "opaque" conditions, whose expression c is treated like a black box.

In *assignment-based partitioning*, each CE$_i$ is ultimately responsible for the whole c, but only for some fraction of the updates. For example, two CEs can partition the workload so that one evaluates c on the even updates, while the other on the odd updates. More formally, each CE$_i$'s condition partition c_i takes on the form $c_i = p_i \wedge c$, where p_i is an "assignment test" on the updates. Intuitively, when a new update arrives at CE$_i$, the assignment test is performed first. If the test is passed, we say that the new update has been *assigned* to CE$_i$. In this case, processing of the update continues as in the case of a centralized system. On the other hand, if the assignment test fails, CE$_i$ does not even evaluate c (in other words, the conjunction in $p_i \wedge c$ is short-circuited). In this way, each CE$_i$ is only responsible for (i.e., evaluates and generates alert for) those updates assigned to it. Thus, load balancing is achieved by controlling the fraction of updates that are assigned to each CE$_i$.

To illustrate the benefit of partitioning, let us assume that processing a new update takes exactly one unit of time in a centralized system monitoring condition c. Such a system can handle a "Maximum Sustained Update Rate" (defined more formally in Section 5.1) of 1 new update per unit time. Further assume that CE$_i$ only takes $\frac{1}{N}$ time unit on average to process an update, either because c_i is a much simpler expression, or because most of the updates received are not assigned to CE$_i$ and can be disregarded quickly. Consequently, such a system can withstand a Maximum Sustained Update Rate of roughly N updates per time unit, resulting in an N-fold increase in capability.

Another benefit of a partitioned system is increased partial reliability of the monitored condition. Since the condition is monitored by several CEs together, even if one of them goes down, the user should still be able to receive some alerts, unlike in the centralized case. In a previous paper [2], we considered full replication as a solution for reliability. In a replicated monitoring system (Figure 1(c)), multiple CEs all monitor the same condition to guard against failures. However, the Maximum Sustained Update Rate of such a system is not improved compared to a centralized system, even though more CEs are deployed. A partitioned system is able to reap some of the benefits of a replicated system without its full cost.

This paper addresses the problem of partitioning a condition so that it can be handled by multiple CEs in parallel. In particular,

- We define a set of desirable properties that contribute to making a partitioned system "equivalent" to a centralized system (Section 3.1).
- We prove some fundamental properties of partitioned systems in general (Section 3), and assignment-based partitioning in particular (Section 4).
- We propose and compare two methods of doing assignment-based partitioning. We also present a few representative systems, and develop performance metrics to measure their relative merits (Section 5).
- Finally, we develop methods to apply our analysis to more complex system configurations involving multiple variables (Section 6).

2 Problem Specification

In this section, we give more details on the workings of a condition monitoring system, using the nuclear reactor temperature sensing example from Section 1. The Data Monitor is a temperature sensor attached to the reactor. It is also connected to a communications network which allows it to send temperature readings to other devices. We assume that each DM monitors only one variable, as a sensor which simultaneously monitors two targets can be thought of as two DMs co-located on the same device.

An update has the format $u(varname, data, seqno)$ where $varname$ is an identifier of the real world variable being tracked, and $data$ reports the new state of this variable. The $seqno$ field uniquely identifies this update in the stream of updates from the same variable. We assume that sequence numbers of updates sent from the same DM are consecutive. In our reactor example, an update $u(x, 3000, 7)$ denotes the seventh update sent by this DM for reactor x, reporting a temperature reading of 3000 degrees. In the remainder of this paper, we will use 7^x to denote such an update.

A condition c is a predicate defined on values of real world variables. The set of variables that appear in a condition's predicate expression is the *variable set* of that condition, denoted by V. The CE receives updates from all DMs that monitor variables in V. When a new update arrives, the CE re-evaluates its condition. The update is said to match (or trigger) c if the data contained in it causes c's predicate to evaluate to true. For example, condition $c1$ ("reactor temperature is over 3000 degrees") is triggered whenever the temperature reading exceeds 3000.

Note that to evaluate condition $c1$, only the current temperature reading is needed. However, to monitor another condition $c2$ ("reactor temperature has risen for more than 200 degrees since last reading"), the CE needs to remember the previous update in addition to the current one. Thus, we generalize to say that a condition is defined on a set (H) of "update histories," one for each variable in V. An *update history* for variable x, denoted H_x, is a sequence of D x-updates received by the CE. Specifically, $H_x = \langle H_x[0], H_x[-1], H_x[-2], \ldots, H_x[-(D-1)]\rangle$, where $H_x[-i]$ is the ith most recently received update of variable x. (See later for how to choose D). When a new x-update is received, it is first incorporated into H_x, which is then used to evaluate the condition. For instance, after update 7^x arrives, $H_x[0]$ becomes 7^x, and $H_x[-1]$ is 6^x, and so on.

The number D, called the degree of H_x, is determined by the condition. We say that a condition c is of degree D with respect to variable x if the evaluation of c needs an H_x of at least degree D. The degree of a condition is inherent in the nature of the condition itself, and it dictates how many x-updates the CE will need to store locally (i.e., the degree of H_x). Thus, condition $c1$ can be expressed more formally as $c1(H) = (H_x[0].data > 3000)$, and is of degree 1 to variable x. On the other hand, $c2(H) = (H_x[0].data - H_x[-1].data > 200)$, and is of degree 2.

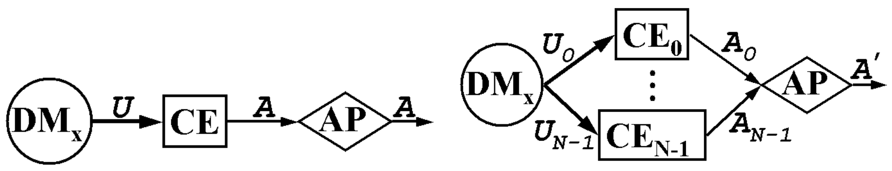
When the condition evaluates to true, an alert is sent out by the CE. An Alert Presenter collects such alerts and displays them to the end user, e.g., by a pop-up window or an audible alarm. We assume that the alert is of the form $a(condname, histories)$, where $condname$ identifies the condition that was triggered, and $histories$ are all the update histories used by the CE in evaluating the condition. The histories are included if the AP needs them for a final round of processing on the alerts before presenting the alerts to the user, as will become clear later.

In a partitioned system, N CEs (CE$_0$ through CE$_{N-1}$) work in concert to handle a single condition c. Instead of c, each CE$_i$ monitors a modified *condition partition* c_i. When a new update arrives at CE$_i$, an alert is triggered if and only if $c_i(H)$ is true. Although the CEs monitor their respective condition partitions independently, the goal is to produce an "equivalent" overall effect to a single CE monitoring the original c.

Finally, we assume that the links connecting the CE to the DMs and APs provide ordered and lossless delivery. The focus of this paper is not on reliability, and the reader is referred to [2] for discussion of issues when the links assumption does not hold.

3 Single Variable Conditions

In this section, we present some general results of partitioned systems, regardless of whether the structure-based or assignment-based method is being used. For now, we restrict our discussions to conditions involving only one real world variable x, i.e., $V = \{x\}$. Hence, the monitoring system contains only one Data Monitor, and all relevant updates will have x as their $varname$. Single variable systems are important both as basis for more complex configurations, and in their own right. Many useful real-world scenarios, such as single-issue stock tracking and danger alerts, can be formulated as a single variable monitoring problem. In Section 6 we will investigate multiple DMs.



(a) Centralized system $\mathcal{C}$ (b) Partitioned system $\mathcal{P}$

Fig. 2: Analysis model for single variable systems

Figure 2(a) depicts a centralized condition monitoring system $\mathcal{C}$. Let U represent the sequence of updates sent out by the DM over some period of time. The Condition Evaluator receives the sequence of updates U as input, and generates alert sequence A as output, according to the definition of c, the condition it is monitoring.

Figure 2(b) illustrates a partitioned one-variable system $\mathcal{P}$ based on the centralized system $\mathcal{C}$. For now, we assume that all CEs in $\mathcal{P}$ receive the same input as the single CE in the corresponding centralized system $\mathcal{C}$, i.e., $U_0 = U_1 = \cdots = U_{N-1} = U$. In other words, the update stream is fully replicated to all CEs. Note that replicating the update stream N times will likely mean more work for the DM. If it is desired to keep DM itself simple (since it can be a dumb device such as a networked thermometer), an additional “update replicator” can be installed to receive updates from the DM and forward them to all CEs. Section 4 will also investigate scenarios where the full update replication assumption can be relaxed.

Since each CE$_i$ monitors a different c_i, their outputs will be different. Specifically, we use A_i to denote the sequence of alerts generated by CE$_i$. Finally, the AP collects all A_i’s and merges them to produce a final alert sequence A', which are displayed to the end user. To produce A', the AP uses a simple duplicate elimination algorithm called ExactDuplicateRemoval to filter out some alerts. In particular, the AP discards an alert if it is “identical” to another alert that has already been presented to the user. Two alerts are considered identical if their history sets H are the same, i.e., if they triggered on the same set of updates.

3.1 System Properties

We propose three desirable properties of a partitioned system, defined by comparing the output of $\mathcal{P}$ to that of $\mathcal{C}$. We define an alert a’s sequence number to be $H_x[0].seqno$, namely, the sequence number of the x-update that triggered a. A sequence A of such alerts is *ordered* if the sequence numbers of all alerts contained in A form a ordered sequence. Furthermore, we use ΦA to denote the (unordered) bag consisting of these sequence numbers.

A partitioned system $\mathcal{P}$ is said to have each of the following properties if every alert sequence A' it outputs satisfies the corresponding criterion.

1. **Orderedness:** A' is ordered.
2. **Consistency:** $\Phi A' \subseteq \Phi A$.
3. **Completeness:** $\Phi A' = \Phi A$.

The three properties measure how the behavior of $\mathcal{P}$ "conforms" to that of $\mathcal{C}$. Specifically, orderedness looks at the order in which alerts are presented to the user, while the other two criteria deal with what alerts are presented. Orderedness indicates that alerts are delivered to the user in increasing sequence number order. Since a centralized system $\mathcal{C}$ always delivers alerts in this order, a partitioned system $\mathcal{P}$ that is ordered behaves similarly in this respect.

If a partitioned system $\mathcal{P}$ is consistent, the user can expect to receive (although perhaps in a different order) a subset of those alerts that would have been generated by the corresponding centralized system $\mathcal{C}$. In contrast, an inconsistent system is able to generate "extraneous" alerts that one would not normally expect from $\mathcal{C}$. Therefore, it is easy for a user behind an inconsistent system to tell that partitioning is being used when he/she sees these "extraneous" alerts.

Completeness is a stricter criterion than consistency. For a partitioned system $\mathcal{P}$ to be complete, it will have to generate all alerts and only those alerts that would have been generated by $\mathcal{C}$. Trivially, completeness implies consistency, while the reverse is not true. An incomplete system implies that the user may miss some alerts as a result of partitioning the workload among several CEs.

3.2 Discussion

How a condition c is partitioned among the CEs (specifically, the definition of c_i's) largely determines the properties achieved by the resulting system $\mathcal{P}$. The following theorems discuss the details. Due to lack of space, we omit all proofs in this paper. Bear in mind that the discussion of this section pertains to single variable conditions, with ExactDuplicateRemoval filtering, and where every CE$_i$ sees the same input.

Theorem 1. *A partitioned system $\mathcal{P}$ is ordered if and only if $\nexists U$, such that $c_i(H) = c_j(H') = true$ and $c_j(H) = false$ for some $i, j \in [0..N-1], i \neq j$ and $H, H' \in U, H_x[0].seqno < H'_x[0].seqno$.*

We deduce from Theorem 1 that in most cases orderedness is not achieved in a partitioned system. This is because normally, when the output alert sequences of different CEs are merged together at the AP, it is possible that alerts will be delivered out of order due to the different interleaving order of these sequences. Exceptions do exist, but often involve an impractical "trivial" partitioning. An example of such a trivial partitioning is where $c_0 = c$ and $c_i \equiv false$ for all other c_i's.

If orderedness is desired, however, it can be enforced by having the AP perform additional filtering on top of ExactDuplicateRemoval. For example, the AP can remember the last alert it has delivered to the user, and discard new alerts that arrive out of order. However, orderedness is gained in this situation at the expense of throwing out some potentially useful alerts.

Theorem 2. *A partitioned system $\mathcal{P}$ is consistent iff $c_0 \vee c_1 \vee \cdots \vee c_{N-1} \Longrightarrow c$.*

Intuitively, if $c_0 \vee c_1 \vee \cdots \vee c_{N-1}$ logically implies c, then any update that triggers at some CE$_i$ in $\mathcal{P}$ will also trigger condition c in $\mathcal{C}$. Analogously, the following theorem states that $\mathcal{P}$ is complete if and only if $c_0 \vee c_1 \vee \cdots \vee c_{N-1}$ and c are logically equivalent.

Theorem 3. *A partitioned system $\mathcal{P}$ is complete iff $c_0 \vee c_1 \vee \cdots \vee c_{N-1} \iff c$.*

Example 1. Condition c is defined as “temperature is over 3000 degrees OR temperature has risen for more than 200 degrees”, or $c(H) = (H_x[0].data > 3000 \vee H_x[0].data - H_x[-1].data > 200)$. Splitting the disjunction using structure-based partitioning, we obtain $c_0(H) = (H_x[0].data > 3000)$, and $c_1(H) = (H_x[0].data - H_x[-1].data > 200)$. Applying the theorems above, we can prove that $\mathcal{P}$ is complete, but unordered.

4 Assignment-Based Partitioning

In this section and the next, we focus on assignment-based partitioning. We have $c_i(H) = p_i(u) \wedge c(H)$, where p_i is the assignment test defined on the newly received update u. We assume that evaluating $p_i(u)$ is a fast operation, with negligible cost, compared to evaluating $c(H)$. This ensures that the goal of partitioning is achieved, namely, that the average workload at each CE$_i$ is reduced compared to the single CE in a centralized system, as long as CE$_i$ is assigned only a portion of the steam of updates in U.

4.1 Discussion

The following theorems, derived from those in Section 3.2, illustrate how the definition of p_i determines the properties of an assignment-based system $\mathcal{P}$. As such, the theorems apply to single variable conditions, with EXACTDUPLICATEREMOVAL filtering, and where every CE$_i$ sees the same input.

Theorem 4. *An assignment-based partitioned system $\mathcal{P}$ is ordered if and only if $\nexists U$ such that $p_i(u) = p_j(u') = true$ and $p_j(u) = false$ for some $i, j \in [0..N-1], i \neq j$ and $u, u' \in U, u.seqno < u'.seqno$.*

Theorem 5. *An assignment-based partitioned system $\mathcal{P}$ is always consistent.*

Theorem 6. *An assignment-based partitioned system $\mathcal{P}$ is complete if and only if $\forall u$, $\mathbf{C}_{i=0}^{N-1}(p_i(u) = true) \geq 1$, where $\mathbf{C}_{i=0}^{N-1}(p_i(u) = true)$ counts the number of i's in $[0..N-1]$ that make $p_i(u)$ true.*

Theorem 6 says that $\mathcal{P}$ is complete as long as every update is assigned to at least one CE$_i$. We observe that duplication of assignment (i.e., one update assigned to more than one CEs) results in replication of the condition (i.e., the condition being monitored by multiple CEs simultaneously), which may actually be beneficial in terms of reliability. Although replication does not affect the system properties (because duplicate alerts are eventually removed by the AP), it has its own issues which are not studied here. Instead, we will focus on systems where each update is assigned to exactly one CE$_i$.

4.2 Types of Assignment

Assignment-based partitioned systems are further divided into two categories depending on how the assignment is determined. In *autonomously assigned* systems, the assignment is done by the CEs autonomously. The CEs agree on the definitions of p_i's before the system starts running. The DM, not knowing these definitions, simply replicates all updates to all CEs concerned.

A *centrally assigned* partitioned system, on the other hand, relies on a central control to decide on the assignment at run time. Before an update u is sent out, the DM inserts an $aceid$ tag, which contains the id of the CE that this update is assigned to ($0 \leq aceid < N$).[1] At the other end, CE$_i$ recognizes that an update has been assigned to it if the $aceid$ matches its own id. In effect, $p_i(u) = (i = u.aceid)$.

While Theorems 4 to 6 above apply to both types of assignment-based systems, the following lemmas are tailored specifically for a single variable partitioned system with central assignment. Such a system is not ordered except in the "trivial" case where all updates are assigned to one particular CE$_i$. It is also always complete because it assigns each update to exactly one CE.

Lemma 1. *A centrally assigned partitioned system $\mathcal{P}$ is ordered if and only if $\exists k \in [0..N-1]$ such that $\forall u, u.aceid \equiv k$.*

Lemma 2. *A centrally assigned partitioned system $\mathcal{P}$ is always complete.*

Central assignment can potentially avoid some shortcomings of an autonomously assigned system. For example, when a new CE joins or an existing CE leaves an autonomously assigned system, all other CEs need to be notified in order to redistribute workload to achieve balance. This is because the definition of p_i usually depends on N, the total number of CEs. Hence when N changes, all p_i's must be redefined. Analogously, when one CE goes down or gets overloaded temporarily, it is hard to dynamically adjust the work distribution to adapt to the change without inter-CE communication. With central assignment, on the other hand, the exact assignment of an update u is finalized just before it is sent out (when $aceid$ is tagged on). Thus, a centrally assigned system has much more flexibility in adapting to a dynamic environment.

4.3 Dropping Updates

Another problem with autonomous assignment is that the update stream U must be fully replicated N times ($U_0 = U_1 = \cdots = U_{N-1} = U$). Because the DM does not know about the definition of p_i, it has no way of predicting which CE or CEs will need a particular update. Hence it must send any update to all CEs to make the system work. Without an efficient multicast protocol, this duplication of update sends can be quite wasteful.

In contrast, as an optimization of a centrally assigned system, the DM can potentially drop some updates to certain CEs in order to avoid unnecessarily sending them. Since the DM controls update assignment, it is also in a position to predict whether a particular update u needs to be sent to a particular CE$_i$ in order for the system to function. In essence, we can reduce U_i, the input to CE$_i$, to a particular subsequence of U, instead of the full U, while keeping the system outcome the same.

One might think that only assigned updates need actually be sent to CE$_i$ (in other words, $U_i = \langle u \mid u \in U \text{ AND } u.aceid = i \rangle$). However, as the following example shows, this naive approach does not work. In fact, the minimal U_i also depends on D, the degree of the condition being monitored.

[1] Note that this requires a more intelligent DM. As noted before, an "update replicator" can be used to keep the DM itself simple. Also, note that the use of a central control in this particular case does not create an additional single point of failure as the DM is needed even with autonomous assignment.

Table 1: Comparison of assignment-based systems. $\Re_i$ is the i-th element of an infinite random sequence of integers, with each element between 0 and $N-1$, inclusive

Name	**Definition**	**Comp.**	**Ord.**	**MSUR**	**AWT**	**PUD**
TRIVIAL	0	$\surd$	$\surd$	1	0	$1-\frac{1}{N}$
RANDOM	$\Re_{u.seqno}$	$\surd$	X	$N-\epsilon$	$2(\frac{N}{\epsilon}-1)$	$(1-\frac{1}{N})^D$
ROUNDROBIN	$u.seqno \bmod N$	$\surd$	X	N	0	$\max(0, 1-\frac{D}{N})$
q-RNDRBN	$\lfloor\frac{u.seqno}{q}\rfloor \bmod N$	$\surd$	X	N	$\frac{(N-1)(q-1)}{2}$	$\max(0, 1-\frac{q+D-1}{qN})$

Example 2. Assume a condition of degree 2: "reactor temperature has risen for more than 200 degrees since last reading". Also assume that update 7^x is assigned to CE$_1$, but 6^x is not. If $U_i = \langle u \mid u \in U$ AND $u.aceid = i\rangle$, then $7^x \in U_1$, but $6^x \notin U_1$. However, in order for CE$_1$ to correctly evaluate the condition when it receives 7^x, it will also need the data from 6^x. Therefore, even though 6^x is not assigned to CE$_1$, it must still be sent to it.

The following lemma shows precisely when an update can be dropped. In short, u is sent to CE$_i$ if it is assigned to CE$_i$, or if a later update (u_k) will be assigned to CE$_i$ and the condition evaluation of u_k depends on u.

Lemma 3. *The output of a centrally assigned partitioned system remains the same if U_i is changed from U to $\langle u \mid u \in U$ AND $\exists u_k \in U$ such that $u_k.aceid = i$ and $0 \leq u_k.seqno - u.seqno < D\rangle$.*

5 Sample Assignment-Based Systems

In this section we show a few sample ways of constructing an assignment-based partitioned system $\mathcal{P}$. For each system, both an autonomously assigned version and a centrally assigned version exist, and they behave identically (except that the latter can drop certain updates as an optimization).

Table 1 summarizes the major results which we explain next. If the definition of a system is given as $\mathbf{d}$ in the table, then the autonomously assigned version is obtained by $p_i(u) = (i = \mathbf{d})$, while the centrally assigned version is defined as $p_i(u) = (i = u.aceid)$ and $u.aceid = \mathbf{d}$.

5.1 Performance Metrics

In order to quantitatively compare different systems, we develop three performance metrics to measure aspects of a system such as its throughput and mean response time. We use a relatively simple analysis model to capture the important system tradeoffs. Our model assumes that updates in U are generated at a constant rate of α. For example, if $\alpha = 2$, then a new update appears every half time unit. When a new update arrives at a CE, the quick assignment test is performed first. If assigned, the update is processed by this CE. Processing time is assumed to take 1 time unit exactly. However, if the CE is currently busy processing another previous update, the new update is appended to an update queue at this CE.

Our first metric measures the throughput of the overall system. The *Maximum Sustained Update Rate* (MSUR) of a partitioned system is the maximum α at which the system is stable, i.e., the update queues at all CEs reach a steady state. As a reference, a centralized system $\mathcal{C}$ has an MSUR of 1 (update/time unit). At update rates greater than 1, updates are generated at a higher pace than they can be consumed by the CE, and the queue length increases without bound.

The *Average Wait Time* (AWT) is the average time an update has to spend waiting in the update queue, while the system is running at MSUR. AWT measures the average response time of the system. In the centralized example, the AWT is 0 (time unit) because a new update arrives just when the previous one finishes processing.

Finally, the *Percentage of Updates Dropped* (PUD) metric gives the average percentage of updates dropped in the centrally assigned version of a partitioned system. For example, if the centrally assigned version only needs to send each update to half of the N CEs on average, then the system has a PUD of 50%. The larger the PUD number, the less work the central control does. If PUD $= 0$, the centrally assigned solution does not save any effort over autonomous assignment.

5.2 Comparison

Due to space limitations, we will only present here a brief comparison of systems, and omit details such as derivations of various results summarized in Table 1. Our first sample system, named TRIVIAL (Figure 3(a)), is a "trivial" system because it assigns all the work to one particular CE, CE_0. In the second system RANDOM (Figure 3(b)), an update is assigned to a random CE every time. Note that to implement the autonomously assigned version of RANDOM, the CEs simply need to agree on a pseudo-random algorithm and a seed at the beginning in order to avoid communication during run time. ROUNDROBIN (Figure 3(c)) assigns each update to the next CE in turn. Finally, q-ROUNDROBIN is a variation on regular ROUNDROBIN where each CE gets assigned q consecutive updates in U before next CE's turn. Naturally, ROUNDROBIN is simply a special case of q-ROUNDROBIN with $q = 1$.[2]

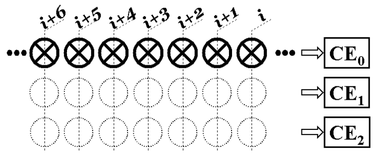
Figure 4 plots PUD against N, the total number of CEs, based on the formulas in Table 1. The various curves represent several different sample systems. The figure shows that in general PUD rises with more CEs, implying that the advantage of central assignment in dropping updates is more significant. Incidentally, TRIVIAL has the best PUD among all partitioned systems because the DM in a centrally assigned TRIVIAL only needs to send updates to one CE, CE_0. However, since one CE handles all the real work, the system obviously does not benefit from being a partitioned system. As such, TRIVIAL has the same MSUR (1) as a centralized system $\mathcal{C}$.

Since RANDOM distributes the work more evenly among the N CEs, it is able to significantly improve its throughput compared to TRIVIAL. Its MSUR is $N - \epsilon$, where ϵ is an arbitrarily small positive number.[3] Note that its MSUR can get infinitesimally close to N, but not equal to N, due to the randomness in how often each CE gets assigned. Moreover, as MSUR approaches N, the AWT suffers as a result.

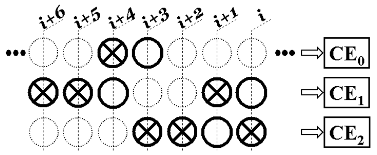
Because of its regular assignment pattern, ROUNDROBIN further improves on RANDOM with an MSUR of N and no wait time for the updates (AWT $= 0$). On the other

[2] In fact, TRIVIAL can also be considered as a special case of q-ROUNDROBIN with $q \to \infty$.

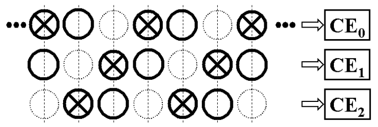
[3] The calculation assumes Poisson random arrival at each CE_i (M/D/1 queue), which is a close approximation especially for small time units.



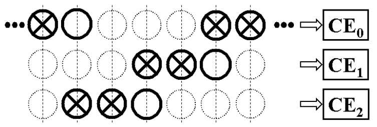
(a) TRIVIAL



(b) RANDOM



(c) ROUNDROBIN



(d) q-ROUNDROBIN

Fig. 3: Running illustrations of sample partitioned systems, with $N = 3$, $D = 2$, and $q = 2$. A crossed circle (⊗) means that an update is assigned to a particular CE. A circle (◯) implies that the update is not assigned but still must be delivered to this CE. A dotted circle can be dropped by a centrally assigned system

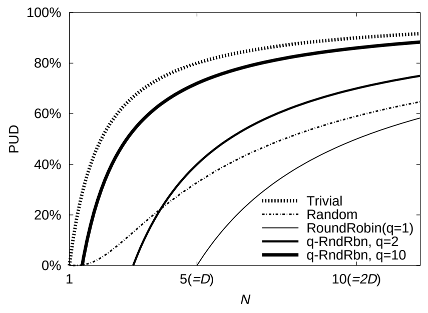


Fig. 4: Comparison of PUD performances

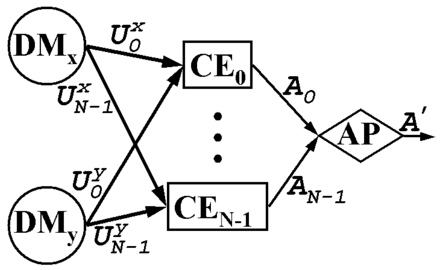


Fig. 5: A partitioned system monitoring a condition with two variables: x and y

hand, as Figure 4 shows, ROUNDROBIN's PUD curve falls below that of RANDOM, especially for small N's. The figure also points out that there is a threshold to N ($N = D$) below which no updates can be dropped by ROUNDROBIN.

Finally, q-ROUNDROBIN is designed to improve ROUNDROBIN's PUD performance, while preserving its MSUR at N. As shown in Figure 4, a bigger q enhances the PUD by both reducing the N threshold and increasing the rate at which the curve rises with N. For instance, at $N = 2D$, an average update can be dropped to half (50%) of the CEs if $q = 1$, but almost 70% if $q = 2$.

The tradeoff of a bigger q, on the other hand, is that the stream of updates assigned to a particular CE$_i$ will be more "bursty", i.e., a bigger chunk of consecutive update assignment, followed by a longer inactivity period. As a result, updates have to wait longer in the queue on average. For example, with 10 CEs, regular ROUNDROBIN (i.e., $q = 1$) has no delay in processing updates, whereas an update has to wait on average 18 time units when $q = 5$. Hence, q represents a tradeoff between savings in dropped updates and faster system response time.

6 Multi-variable Conditions

So far we have dealt with conditions involving only a single variable. In this section we study conditions whose variable set contains more than one variable, i.e., $|V| > 1$. To illustrate, Figure 5 shows a system with two independent data sources, x and y. An example of such a condition is "temperature difference between two reactors x and y exceeds 100 degrees."

Some of the results in previous sections can be naturally extended to the multi-variable case. For example, we can define orderedness, consistency and completeness for multi-variable systems by extending the definitions of Section 3.1 in a straightforward manner. Similarly, performance metrics for multi-variable systems can also be defined. The detailed definitions are omitted here to avoid redundancy. However, additional complications do arise in a multi-variable system, which we explore next.

6.1 Discussion

As in the single variable case, it can be shown that orderedness is seldom achieved in multi-variable systems except for some trivial configurations. Therefore, we will focus only on consistency and completeness in the following discussion. For now, we assume that no updates are dropped (in Figure 5, $U_0^x = U_1^x = \cdots = U_{N-1}^x$, and likewise for y). We will explore the interesting issue of dropping updates later in this section.

From Figure 5, we observe that the input to CE$_i$, U_i, is a mixed sequence of x- and y-updates, obtained from interleaving U_i^x and U_i^y. Without any safeguard mechanism, it is possible for U_i^x and U_i^y to interleave differently at different CEs, resulting in the CEs seeing a different input from each other. Consequently, a multi-variable partitioned system is no longer consistent even if the condition in Theorem 2 is satisfied. The following example illustrates.

Example 3. Assume the condition c being monitored is "temperature difference between two reactors x and y exceeds 100 degrees" ($c(H) = (|H_x[0].data - H_y[0].data| > 100)$). We split c into $c_0(H) = (H_x[0].data - H_y[0].data > 100)$, and $c_1(H) = (H_y[0].data - H_x[0].data > 100)$. Notice that $c_0 \vee c_1 \Longrightarrow c$.

Let $U_0^x = U_1^x = \langle 1^x(1000^\circ), 2^x(1200^\circ)\rangle$, and $U_0^y = U_1^y = \langle 1^y(1000^\circ), 2^y(1200^\circ)\rangle$. That is, both reactors start at 1000 degrees, and then both increase to 1200 degrees. Further assume that streams U_0^x and U_0^y are interleaved at CE$_0$ such that $U_0 = \langle 1^x, 1^y, 2^x, 2^y\rangle$. However, a different interleaving at CE$_1$ makes $U_1 = \langle 1^x, 1^y, 2^y, 2^x\rangle$.

In this case, CE$_0$ triggers when it receives 2^x, and CE$_1$ will also trigger when it receives 2^y. The user will receive both alerts (both will pass through the AP's filtering system since they are not considered duplicates). However, one can prove that no centralized monitoring system could generate such an alert combination. Hence the partitioned system violates consistency.

There are two general approaches to remedy the above consistency problem in multi-variable systems. First, measures can be taken to ensure that each CE sees the same input. For example, a centralized "update replicator" can receive updates from all DMs involved in a condition and then forward them to the CEs checking that condition, as hinted in earlier sections. As another example, a physical clock system as defined in [3] can be put in place to enforce a total ordering of x- and y- updates at all CEs.

In the second approach, the CEs are allowed to see different input. However, the Alert Presenter is required to take on a more active role in filtering out alerts that can potentially lead to inconsistency, as illustrated in the next example. The disadvantage is that the AP may potentially discard useful alerts.

Example 4. We assume the same setup as in Example 3. Furthermore, each alert sent to the AP is tagged with the updates that it triggered on. For example, the alert generated by CE$_0$ will be tagged with $\{2^x, 1^y\}$. The AP records this information before passing the alert on to the user. When the second alert arrives from CE$_1$ tagged with $\{1^x, 2^y\}$, the AP will be able to detect inconsistency (algorithm detail omitted due to space constraint) and thereby discard the new alert.

In the rest of our discussion, we assume that all CEs see the same input. In fact, the following theorem tells us that, with such an assumption, all earlier single-variable results on consistency and completeness become valid for multiple variables as well.

Theorem 7. *Given $U_0 = U_1 = \cdots = U_{N-1}$, Theorems 2, 3, 5 and 6, and Lemma 2 apply to multi-variable partitioned systems.*

6.2 Dropping Updates

Another interesting consequence of multiple variables is that the DMs in a centrally assigned system can no longer drop updates as freely. Previously in the single variable case, the DM for variable x controls the assignment of x-updates, and it can accurately predict whether a particular x-update u will not be needed by CE$_i$ in its condition matching. With multiple independent DMs, however, such a prediction can no longer be safely made in general because, even when DM$_x$ considers u unnecessary for CE$_i$, u might still be needed in evaluating the condition when a later y-update is sent by DM$_y$.

Example 5. Given a two-variable centrally assigned system, assume that a certain x-update, 6^x, is not assigned to a particular CE$_i$. Further assume that the condition is of degree 1 to variable x, i.e., only the most recent x value is needed in condition evaluation. Thus, it would appear that 6^x can be safely dropped by DM$_x$ to this CE.

The problem comes, however, when a y-update, say, 3^y, arrives next. Assume 3^y is assigned to this CE. To evaluate the condition, the CE needs the most recent value of x, which should have been in 6^x. However, since the CE never received 6^x, value from an earlier 5^x is used instead. Thus the output of this CE is potentially affected by the dropping of 6^x. In fact, the resulting system is no longer consistent.

In the most general case it is difficult for the DMs to safely drop an update without communication between themselves. However, specific circumstances exist where it is possible to do so, and the following lemma presents one such scenario. In essence, Lemma 4 says that an x-update can be dropped to CE$_i$ if no updates of any variable other than x are assigned to CE$_i$, and if the update will not be used to evaluate conditions triggered by assigned x-updates.

Lemma 4. *An update u can be dropped to CE$_i$ if $\nexists u_k \in U_i$ such that $u_k.aceid = i$ AND ($u_k.varname \neq u.varname$ or $0 \leq u_k.seqno - u.seqno < D$).*

1. Partition (arbitrarily) the set of N CEs into $|V|$ disjoint and non-empty groups, one G_v for each variable $v \in V$.
2. For each G_v, pick any single variable partitioning scheme (such as those presented in Section 5), which will be used to assign v-updates to CEs in G_v.
3. For each update u,
   3-1. Assign u to one of the CEs in $G_{u.varname}$, according to the group's chosen single variable scheme. We thus have CE$_{u.aceid} \in G_{u.varname}$.
   3-2. Then, send u to all CEs in all groups G_v where $v \neq u.varname$, and in addition, to the subset of CEs in $G_{u.varname}$ as dictated by the single variable scheme.

Fig. 6: Algorithm MVP. For simplicity we assume that $N \geq |V|$

Based on the above lemma, we have developed an algorithm (Figure 6) to perform the partitioning in a centrally assigned multi-variable system. Algorithm Multi-Variable-Partitioning (MVP) associates a subset of CEs to each variable in the condition. It then reduces an overall multi-variable problem to single variable partitioning problems within each variable subset. To illustrate how the algorithm works, we walk through a simple example next.

Example 6. Assume a system with four CEs and two variables. For simplicity, further assume that the condition is of degree 1 to both x and y. Using Algorithm MVP, let $G_x = \{\text{CE}_0, \text{CE}_1\}$ and $G_y = \{\text{CE}_2, \text{CE}_3\}$. DM$_x$ decides to use ROUNDROBIN assignment within its group, while DM$_y$ uses RANDOM.

Based on ROUNDROBIN, all x-updates with odd sequence numbers are assigned to CE$_1$, while even ones are assigned to CE$_0$. In other words, $\forall u$ with $u.varname = x$, we let $u.aceid = u.seqno \bmod |G_x| = u.seqno \bmod 2$. Furthermore, odd sequence numbered updates are sent to CE$_1$, its assigned CE, as well as to both CEs in G_y. Analogously for even sequence numbered x-updates. Similarly, a y-update is assigned randomly to either CE$_2$ or CE$_3$ according to RANDOM. Then it is sent to its assigned CE in G_y, plus to CE$_0$ and CE$_1$.

Intuitively, an even numbered x-update u can be safely dropped to CE$_1$ because it is not needed to evaluate the condition when CE$_1$ receives an assigned odd numbered x-update. Furthermore, since CE$_1$ is never assigned any y-updates, it does not have any other evaluations which could have required u. Finally, CE$_1$ is guaranteed to have the correct y-value in its evaluations since it still receives all y-updates. Because each update is sent to exactly three out of four CEs, the system has an overall PUD of 25%.

As Theorem 8 shows, Algorithm MVP results in systems that are guaranteed to be complete, while still allowing updates to be systematically dropped. We further observe that the performance of the overall system is closely tied to that of the single variable schemes selected in Step 2, and a detailed analysis is omitted due to space limitations.

Theorem 8. *A centrally assigned multi-variable partitioned system produced by Algorithm MVP is complete as long as each single variable scheme selected in Step 2 generates complete systems.*

7 Related Work

Modern content-based publish/subscribe systems [4–6] route and filter events from their sources to interested destinations using a network of servers. There is certain overlap of functionality between these systems and the condition monitoring systems we are interested in. However, the focus of this work is on partitioning a condition to be handled by multiple servers while maintaining the same semantics.

SIFT [1] implemented an earlier monitoring system that provided batched filtering of newsgroup articles. Based on its experience in operation, SIFT pointed out and motivated the need for workload distribution. It also proposed the SIFT Grid, which is a distributed mechanism resembling a variation of our centrally assigned scheme. However, the SIFT Grid only dealt with degree 1 conditions, and properties about the partitioning were never analyzed formally.

Large scale World Wide Web servers often use dynamic load balancing and failover for increased capacity and availability [7]. Although that research mostly focuses on dynamic fault detection and traffic redirection for stateless servers, some of the techniques can very well be adapted to augment our centrally assigned monitoring systems.

Data stream systems are a recent research topic that has generated lots of interest and activities [8]. Although such systems also deal with streams of updates, the direction taken (e.g., data models for continuous queries) is quite different from ours. However, some of our techniques and analysis can very well be applicable in that context.

8 Conclusion

As real-time monitoring of sensors and information sources becomes more widespread, it will be critical to deal efficiently with large volume of updates and complex conditions. Condition partitioning allows multiple servers (CEs) to share the load, but can potentially lead to undesirable outcomes. In this paper we have studied what is needed to preserve orderedness, consistency and completeness in partitioned systems. With assignment-based partitioning, we have shown that CEs have a more balanced load because they handle the same condition over different updates. The metrics and analysis we have presented make it possible to compare assignment-based systems. In particular, our model suggests that ROUNDROBIN performs well in throughput and mean response time, while a centrally assigned q-ROUNDROBIN can trade system response time for less work.

References

1. Yan, T.W., Garcia-Molina, H.: The SIFT information dissemination system. ACM Transactions on Database Systems **24.4** (1999) 529–565
2. Huang, Y., Garcia-Molina, H.: Replicated condition monitoring. In: Proceedings of the 20th ACM Symposium on Principles of Distributed Computing. (2001) 229–237
3. Lamport, L.: Time, clocks, and the ordering or events in a distributed system. Communications of the ACM **21.7** (1978) 558–565
4. Aguilera, M.K., Strom, R.E., Sturman, D.C., Astley, M., Chandra, T.D.: Matching events in a content-based subscription system. In: Proceedings of the 18th Annual ACM Symposium on Principles of Distributed Computing. (1999) 53–61
5. Zhao, Y., Strom, R.: Exploiting event stream interpretation in publish-subscribe systems. In: Proc. of the 20th ACM Symposium on Principles of Distributed Computing. (2001) 219–228
6. Carzaniga, A., Rosenblum, D.S., Wolf, A.L.: Achieving scalability and expressiveness in an Internet-scale event notification service. In: Proceedings of the 19th Annual ACM Symposium on Principles of Distributed Computing. (2000) 219–27
7. Bourke, T.: Server Load Balancing. O'Reilly (2001)
8. Babcock, B., Babu, S., Datar, M., Motwani, R., Widom, J.: Models and issues in data stream systems. In: Proc. of the 2002 ACM Symposium on Principles of Database Systems. (2002)