Replicated condition monitoring

Yongqiang Huang, Hector Garcia-Molina
Department of Computer Science, Stanford University

CONTACT AUTHOR:
Yongqiang Huang
Gates 438
Stanford University
Stanford, CA 94305
yhuang@cs.stanford.edu

Abstract

A condition monitoring system tracks real-world variables and alerts users when a predefined condition becomes true, e.g., when stock price drops, or when a nuclear reactor overheats. Replication of monitoring servers can reduce the probability that an important alert is missed. However, replicated independent servers can sometimes report "conflicting" alerts to the user, causing confusion. In this paper, we study the problem of replicated condition monitoring. We identify and formally define three desirable properties of a replicated system, namely, orderedness, consistency, and completeness. We propose new monitoring algorithms that enforce some or all of the desired properties in different scenarios.

Student status

Regular presentation (Please also consider for Brief announcement if not selected.)

Replicated condition monitoring

Yongqiang Huang, Hector Garcia-Molina
Department of Computer Science, Stanford University

1 Introduction

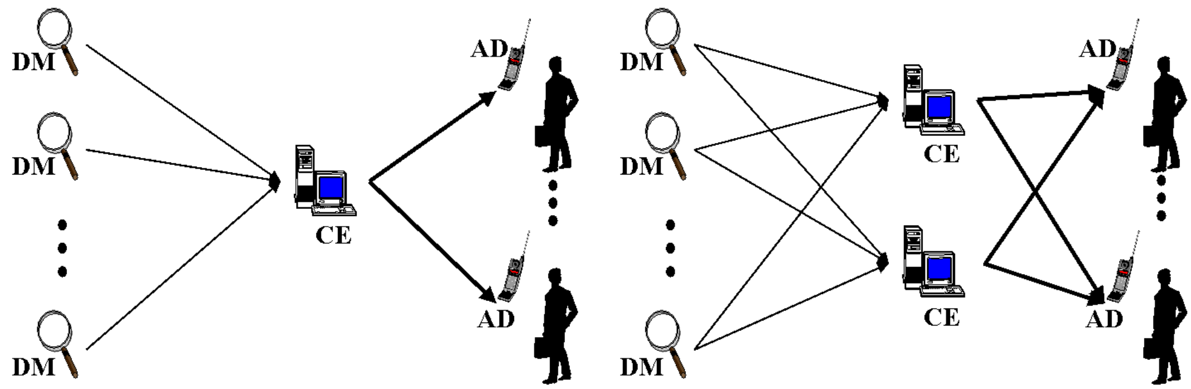
With increasing mobility of today's workforce, there are many scenarios where people must be alerted when certain conditions become true. For example, a traveling business person needs to receive a page whenever the price of a given stock drops below a certain limit. Soldiers in a battlefield must receive an alert at their mobile communication device whenever irregular enemy troop movement is detected by a satellite, or whenever a missile is fired. 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.

For these purposes, a *condition monitoring system* is used. The system monitors real world variables and alerts the user when a predefined condition is satisfied. Figure 1(a) illustrates one such system. It consists of one or more **Data Monitors** (DM), a **Condition Evaluator** (CE), and one or more **Alert Displayers** (AD). 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 a *data update*, i.e., a temperature reading. These updates arrive at the Condition Evaluator, which uses them to evaluate a predefined condition, e.g., "reactor temperature is over 3000 degrees." If the condition is satisfied, an *alert* is sent to the Alert Displayer, 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 AD becomes available.

A Data Monitor resides near the data source it monitors (e.g., the reactor), while an Alert Displayer resides near the end user (e.g., on a PDA). Typically the Condition Evaluator resides on a separate computer in the fixed network, e.g., on a computer in the control room of the plant. There are three reasons why we may not want to place the CE on the same device as a DM. Firstly, the CE will likely require a fair amount of computing resource for data logging and condition evaluation, while a DM is usually a simple sensor device. Secondly, the data sources can be autonomous and do not allow the users to define their own conditions there. For example, the DM can be a stock trading center giving out stock quotes. Individual investors usually cannot ask these sources to monitor conditions for them. Finally, the condition may involve more than one real world variables, making it impossible for the CE to be co-located with all the DMs. For example, the condition "US stock price drops while Japan price climbs" requires data from two sources. Likewise, the CE should be hosted on a separate device from the AD as well. The PDA can be powered off or disconnected from the network most of the time to conserve battery, making it unsuitable to host the CE, which needs to listen constantly for data updates.

We call systems with a single Condition Evaluator, such as the one in Figure 1(a), *non-replicated* monitoring systems. The problem with a non-replicated system is that the CE can go down, causing it to miss updates. Consequently, the CE may not know when a condition is satisfied. Furthermore, the computer network linking the DMs to the CE (called *front links*), or the CE to the ADs (called *back links*) can also be out of service. Thus even when the CE itself is working, the AD may not be able to contact the CE to receive alerts. If a condition is very important, we certainly want to increase the likelihood that we will be notified promptly whenever it is satisfied.

The vulnerability of a non-replicated system can be alleviated by introducing multiple independent Condition Evaluators (Figure 1(b)). We call such systems *replicated* monitoring systems. The multiple CEs all monitor the same condition and receive data updates from the same DMs. They indepen-



(a) Non-replicated

(b) Replicated

Figure 1: Condition monitoring systems.

dently make their own decisions about when the condition is satisfied, and send alerts to the same AD for eventual display to the end user. The redundancy in the system reduces the probability that a critical alert will not be delivered on time (or at all) to the user.

However, the problem of "consistency" arises with replication. Without any safeguard, the user can receive a sequence of alerts that are confusing or even contradictory. For instance, if an alert is to be sent whenever a missile is fired, having two CEs will likely result in two alerts being sent to the user for every missile fired. Without a mechanism to identify duplicates, the user will get confused about the exact number of missiles fired. As another example, suppose a monitoring system is used to report "sharp price drops" of a given stock, defined as greater than twenty percent drops between two consecutive quotes. At first, two price quotes, of values 100 and 50 respectively, are sent. CE1 receives both quotes and generates an alert a_1. However, CE2 only receives the first quote (100). Later when a third quote comes in with a value of 52, only CE2 generates an alert a_2. Since a_2 is not a duplicate of a_1, both will be reported to the user even if duplicate suppression is being used. The user will be confused about when the drop happened. Even worse, he/she may mistakenly think that there have been two drops in price instead of one. Note that such a scenario would not be possible in a non-replicated system.

This paper addresses the replicated monitoring problem. We identify and define a set of properties desirable in a replicated system (Section 3.1). We study the properties of several different types of systems (Section 3). We then develop mechanisms to enforce some of the properties or trade off one against another (Section 4). We also study which types of systems our proposed algorithms work in. Finally, we extend our analysis to more sophisticated system configurations (Section 5 and Appendix D).

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 broadcast temperature readings to other devices. We assume that each DM monitors only one variable. A sensor which simultaneously monitors two targets can be thought of as two Data Monitors co-located on the same device.

A data update is a tuple $u(varname,\ seqno,\ value)$ where $varname$ is an identifier of the real world variable being monitored; $seqno$ uniquely identifies this update in the stream of updates from the same variable; and $value$ is the "new value." We assume that sequence numbers of updates sent from the same variable are consecutive. In other words, the DM keeps a counter, which is incremented for every update generated. We also assume that $value$ contains a full snapshot of the variable rather than an incremental delta. This way, an individual update can still be of use even if its previous one was lost. Hence, a temperature update includes the temperature of the reactor at the time of the reading, instead of the temperature difference from the previous reading. Additionally, the $value$ field could include extra information such as a timestamp. We will ignore any extra information in the $value$ field since it is not essential to our discussion. In our reactor example, an update $u(x, 7, 3000)$ denotes the seventh update sent by the DM for reactor x, reporting a temperature reading of 3000 degrees. In the remainder of this paper, we will use $7^x(3000)$ to denote such an update, or just 7^x when the actual update values are irrelevant to our discussion.

A condition c is an expression defined on values of real world variables. The expression can only evaluate to true or false. For example, condition $c1$ ("reactor temperature is over 3000 degrees") is satisfied whenever the temperature reading exceeds 3000. Similar to the DM, we assume that one Condition Evaluator monitors a single condition.

The set of variables that appear in a condition expression is the *variable set* of that condition, denoted by V. The CE receives data updates from DMs for all variables in V. When a new update arrives, the CE re-evaluates its condition. If the condition is satisfied, the CE will send out an *alert* to the Alert Displayer for eventual display to the interested user.

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 received"), the CE needs to remember the previous data 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 N x-updates received by the CE. Specifically, $H_x = \langle H_x[0], H_x[-1], H_x[-2], \ldots, H_x[-(N-1)]\rangle$, where $H_x[i]$ is the ith most recently received update of variable x. (See later for how to choose N). When a new x-update is received, it is first incorporated into H_x, which is then used to evaluate the condition. For instance, immediately after update 7^x arrives, $H_x[0]$ will be 7^x, and $H_x[-1]$ will be 6^x provided 6^x was not lost, or 5^x if it was, and so on. When the system is just starting up, and the CE has not received at least N x-updates, H_x is undefined.

The number N, called the degree of H_x, is determined by the condition. We say that a condition c is of degree N with respect to variable x if the evaluation of c needs an H_x of at least degree N. 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 $c1(H) = (H_x[0].value > 3000)$, and is of degree 1 to variable x. On the other hand, $c2(H) = (H_x[0].value - H_x[-1].value > 200)$, and is of degree 2. Note that a condition that uses only $H_x[0]$ and $H_x[-2]$ is of degree 3 to x.

We do not consider all possible types of conditions in this paper, because some are impractical, while others are not amenable to replication. Specifically, we do not consider:

1. conditions of an infinite degree in any variable;
2. conditions requiring state information (other than H; e.g., a high watermark) to be stored at the CE, "current temperature exceeds maximum of all previous readings" being one example;
3. conditions involving any notion of "time," such as "temperature is larger than 3000 and current time is past midnight."

We define a condition to be *non-historical* if it is of degree 1 with respect to all variables in V. Otherwise, the condition is *historical* (since it looks at historical data in addition to the most recent updates). Condition $c1$ is non-historical, while $c2$ is historical.

For a historical condition such as $c2$, there is the additional complication of what to do when an update is lost. For example, assume that update 5^x is received by the CE, but 6^x is lost, possibly due to a network problem. Later when 7^x arrives, we have $H_x[0] = 7^x$ and $H_x[-1] = 5^x$. We give the name *conservative triggering* to the class of conditions that detect the loss of an update, and always evaluate to false in such cases. Formally, a condition c is conservative if, for every variable x in V, c is always false if the sequence numbers of updates in H_x are not consecutive. Otherwise, a condition is *aggressive triggering*. Concretely, $c2$ is aggressive because it does not check the consecutiveness of sequence numbers in H to determine whether there has been a missed update. There is also a conservative variant of $c2$, which we will call $c3$: "reactor temperature has risen for more than 200 degrees since last reading taken at the DM." Formally, $c3(H) = (H_x[0].value - H_x[-1].value > 200)$ AND $(H_x[0].seqno = H_x[-1].seqno+1)$. Whether a condition should be conservative or aggressive depends on the nature of the application.

When the condition evaluates to true, an alert is sent out by the CE. The alert is of the form $a(condname, histories)$, where $condname$ identifies the condition, and $histories$ are all the update histories used by the CE in evaluating the condition. The histories are needed by the Alert Displayer to identify duplicates or conflicts in some cases. Note that, although conceptually we send all histories in an alert, in practice this is often not necessary. As will become clear later on, some systems do not need this information at all. Others need only the update sequence numbers contained in the histories. Still others only use these sequence numbers in a simple equality test, in which case it may be sufficient to send just a checksum of the histories.

Finally, an Alert Displayer collects such alerts and displays them to the end user, e.g., by a pop-up window or an audible alarm. The AD may do a final round of processing on the alerts before presenting them to the user. For instance, the AD may need to suppress duplicate alerts, or it may choose to reorder alerts that arrive out of order. In fact, the filtering algorithm the AD uses has a strong impact on the properties of a system. We will study a few such AD algorithms and their tradeoffs later in this paper.

2.1 Assumptions

We assume that only one condition is being monitored. In Appendix D we look at multiple conditions and the complications that arise. Without loss of generality, we assume a system contains only one AD, as there is no interaction between the ADs in a multi-AD system. For a replicated system, we consider a system with two CEs for simplicity. Analysis for systems with more than two CEs can be easily extended.

We assume that both the *front links* (links carrying updates from the DM to the CE) and the *back links* (links carrying alerts from the CE to the AD) guarantee ordered delivery. Ordered delivery of messages on a link can be obtained easily by having the sender tag all messages with a sequence number, and letting the receiver discard messages that arrive out of order. Such a mechanism is simple to implement, and does not incur much overhead.

We further assume that the front links are potentially lossy while the back links are not. To guarantee lossless delivery over an unreliable physical link requires a reliable communications protocol, such as TCP, which stores and retransmits messages until delivery is confirmed. Such a protocol can incur an overhead that is too costly for the front links for three reasons. First, the Data Monitor may be a simple and low-capability device, such as a networked temperature sensor. Second, the DM may be multicasting updates to many recipients. Third, the data updates may be numerous and continuous. Consequently, it is often more appropriate for the DM to implement a datagram protocol such as UDP which is potentially lossy. On the other hand, a TCP-like protocol is justified on the back links for the following reasons. The overhead is relatively small because much less traffic is expected on the back links, which carry alerts instead of updates. The PDAs are turned off most of the time, so the CE is expected to buffer and store the alerts anyway. Lastly, losing an alert is likely much more undesirable than losing a data update.

2.2 Notation

In our discussions we will be dealing extensively with sequences of natural numbers. We say that such a sequence S is *ordered* if S's elements appear in non-decreasing numerical order. For example, $\langle 3, 8, 100\rangle$ and $\langle 2, 2\rangle$ are ordered sequences, while $\langle 2, 1, 6\rangle$ is not. Furthermore, let ΦS denote the (unordered) set whose elements are those of sequence S. Thus, $\Phi(\langle 2, 1, 2, 6\rangle) = \{1, 2, 6\}$.

Given two sequences S_1 and S_2, we say that S_1 is a *subsequence* of S_2, denoted by $S_1 \sqsubseteq S_2$, if S_1 can be obtained from S_2 by removing zero or more of S_2's elements. And we use $S_1 = S_2$ to denote equality, i.e., $S_1 \sqsubseteq S_2$ and $S_2 \sqsubseteq S_1$.

Lastly, if both S_1 and S_2 are ordered sequences, their *ordered union*, $S_1 \sqcup S_2$, is the ordered sequence that satisfies $\Phi(S_1 \sqcup S_2) = \Phi S_1 \cup \Phi S_2$. For example, if $S_1 = \langle 1, 4, 8\rangle$ and $S_2 = \langle 2, 4, 5\rangle$, then

$S_1 \sqcup S_2 = \langle 1, 2, 4, 5, 8\rangle$. Note that duplicates should be removed from the ordered union.

If U represents a sequence of data updates (i.e., a sequence of u tuples), then $\Pi_x U$ denotes the sequence formed by sequence numbers of x-updates in U. That is, $\Pi_x U = \langle u.seqno \mid u \in U$ AND $u.varname = x\rangle$. For example, given $U = \langle 2^x, 6^y, 1^y, 3^x\rangle$, $\Pi_x U = \langle 2, 3\rangle$, and $\Pi_y U = \langle 6, 1\rangle$. If $\Pi_x U$ is ordered, we say that U is *ordered with respect to variable x*. We will omit the variable name (just ΠU and just say "U is ordered") if it is implied in the context (for example when we are talking about single variable conditions).

Recall that an alert a contains the set of update histories (H) used in evaluating its condition. We define a's sequence number with respect to variable x, denoted $a.seqno.x$, to be $H_x[0].seqno$, namely, the sequence number of the last x-update received when a was triggered. Analogous to U, we define $\Pi_x A$ for a sequence of alerts A as the sequence $\langle a.seqno.x \mid a \in A\rangle$. If this sequence is ordered, we say that A is *ordered with respect to* x. Furthermore, if A is ordered with respect to every variable in V, we simply say that A is ordered.

3 Single variable conditions

In this section, 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 $varname$. In Section 5 we will investigate the impact of multiple DMs.

Figure 2(a) depicts a replicated one-variable system. U represents the sequence of updates sent out by the DM over some period of time. When these updates arrive at CE1 and CE2, they become sequences U_1 and U_2, respectively. Notice that U_1 and U_2 may not necessarily equal U because the front links can be lossy. However, both are subsequences of U because the front links deliver messages in-order. That is, $U_1, U_2 \sqsubseteq U$.[1]

[1]Strictly speaking, U is a sequence of data updates rather than numbers. However, we use U in a context where a natural number sequence is expected with the understanding that each update is represented by its sequence number. Analogously for a sequence of alerts A.

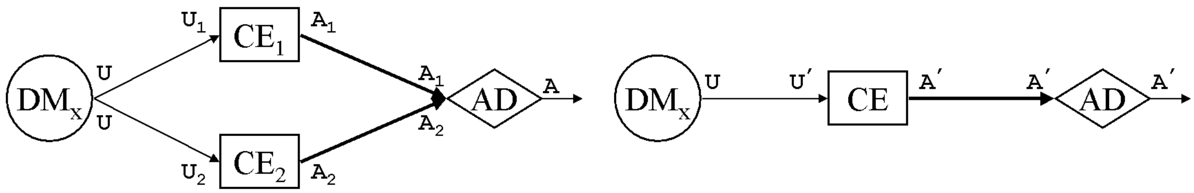
A Condition Evaluator takes in a sequence of updates as input, and generates an alert sequence as output. We use T_c (or just T since c is implicit in a single condition system) to denote the action performed by a CE. Specifically, T is a function that maps a sequence of updates to a sequence of alerts, according to the definition of the condition being monitored. For example, given the input U_1, the output of CE1 will be $A_1 = T(U_1)$. Since the back links are lossless, A_1 will be the sequence of CE1 alerts received at the AD as well.

Finally, the AD collects both A_1 and A_2 and merges them to produce a final alert sequence A, which are displayed to the end user. To produce A, the AD uses an *AD algorithm* to filter out some alerts such as duplicates. In this section, we assume a simple duplicate elimination algorithm (Figure A-1 in Appendix A). Algorithm AD-1 discards one of two "identical alerts" coming from the two CEs. Two alerts are considered identical if their history sets H are the same. For example, assume the condition is of degree 2, and CE1 generates an alert a_1 which triggered on updates 2^x and 3^x, while CE2 generates a_2 which triggered on 1^x and 3^x (because CE2 did not receive 2^x). Although both a_1 and a_2 were triggered when 3^x arrived at their respective CEs, Algorithm AD-1 will not recognize them as "duplicates" because their H are different. Hence both will eventually be reported to the user. There are also other more involved AD algorithms which will be discussed in Section 4.

Example 1 Let us walk through an example in detail. The condition being monitored is $c1$: "reactor temperature is over 3000 degrees." Over time, the DM sends out three updates: $U = \langle 1^x(2900), 2^x(3100), 3^x(3200)\rangle$. All the updates reach CE1 without problem. However, 2^x is lost at CE2. Thus, $U_1 = U$ and $U_2 = \langle 1^x, 3^x\rangle$.

CE1 generates $A_1 = T(U_1) = \langle a_1, a_2\rangle$, where $a_1.H = \langle 2^x\rangle$ and $a_2.H = \langle 3^x\rangle$. CE2 generates only $A_2 = T(U_2) = \langle a_3\rangle$, where $a_3.H = \langle 3^x\rangle$.

Both A_1 and A_2 are sent to the AD, which produces A by filtering out alerts with Algorithm AD-1. Depending on the interleaving order of alerts in the two input streams, A can assume different outcomes. For example, if the order of arrival is a_1, a_3, and then a_2, we will get $A = \langle a_1, a_3\rangle$. That is to say, a_2 is



(a) Replicated system

(b) Corresponding non-replicated system

Figure 2: Analysis model for single variable condition systems.

filtered out by the AD, and two alerts are actually delivered to the user.

3.1 System properties

We propose three desirable properties of a replicated system. A replicated system R is said to have each of the following properties if every alert sequence A it produces satisfies the corresponding criterion.

1. **Orderedness:** A is ordered.
2. **Completeness:** $\Phi A = \Phi T(U_1 \sqcup U_2)$.
3. **Consistency:** $\exists U'$ such that $\Phi A \subseteq \Phi T(U')$ and $U' \sqsubseteq (U_1 \sqcup U_2)$.

To better understand these properties, we define a *corresponding non-replicated monitoring system* N (Figure 2(b)) to a replicated system R as one with otherwise the same configuration as R except that N only has one CE and no filtering is performed at the AD. The three properties measure how the behavior of R "conforms" to that of N. 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 corresponding non-replicated system N always delivers alerts in this order, a replicated system R that is ordered behaves similarly in this respect. However, when R is not ordered, confusion may arise. Imagine a condition which alerts the user of movements in a stock's price. If alerts arrive out of order, the user may be given the false impression that the price is climbing when it is actually dropping, causing the user to make wrong decisions.

One might argue that the above scenario could be prevented by tagging each alert with a timestamp and having the user detect and "disregard" older alerts that arrive out of place. We observe that this solution is very similar to one of the algorithm extensions we will be talking about in Section 4, and we defer the discussion of the tradeoffs involved until then.

If a replicated system R 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 non-replicated system N. On the other hand, an inconsistent system is capable of generating "extraneous" alerts that one would not normally expect from N. Therefore, it is easy for a user behind an inconsistent system to tell that replication is being used when he/she sees these "extraneous" alerts.

Completeness is a stricter criterion than consistency. For a replicated system R to be complete, it will have to generate all alerts and only those alerts that would have been generated by N, when the single CE in N is given the combined inputs of CE1 and CE2 in R (i.e., $U' = U_1 \sqcup U_2$). Trivially, completeness implies consistency, while the reverse is not true. Recall that one of the purposes of replication is to guard against data update loss. A complete system is most effective in this role because it gives the user the maximum set of alerts that can be generated by any consistent system.

To summarize, an ordered and complete replicated system displays exactly the same alerts as its corresponding non-replicated system, and in the same order. An unordered but complete system displays the same alerts, but possibly out of order. An incomplete but consistent system may not be able to display the full set of alerts, but at least it does not generate any "extraneous" alerts. Finally, an inconsistent system is capable of generating such "extraneous" alerts.

3.2 Discussion

The following theorems show which properties are guaranteed under various types of systems. We defer all proofs in this paper to Appendix B. Bear in mind that the discussion of this section pertains to single variable conditions, and Algorithm AD-1.

Theorem 1 (Lossless Links) *A replicated monitoring system with lossless front links and any type of condition is ordered and complete (hence consistent, trivially).*

Because the links are reliable, the advantage of having multiple CEs in such a configuration may not be apparent. One possible benefit is better availability, in the sense that if one of the CEs is temporarily down, the other one can still generate alerts.

Theorem 2 (Non-historical Condition) *A replicated monitoring system with lossy front links and a non-historical condition is complete but not ordered.*

Theorem 3 (Conservative Triggering) *A replicated monitoring system with lossy front links and a conservatively triggered historical condition is consistent, but not ordered nor complete.*

Theorem 4 (Aggressive Triggering) *A replicated monitoring system with lossy front links and an aggressively triggered historical condition is neither ordered nor consistent.*

Table 1 gives a quick summary of the results. Among other things, it shows that the orderedness property is almost never achieved except in trivial cases. The reason is that, when A_1 and A_2 are interleaved and merged at the AD, the result tends to become unordered. In Section 4 we will investigate alternative AD algorithms for situations where orderedness is imperative.

Table 1 also shows that completeness cannot be achieved with historical conditions (except in the trivial case where the front links are lossless). Historical conditions depend on older updates cached at the CE. As a simple example, suppose that update i is received only by CE1, while $i+1$ is received only by CE2. A non-replicated system receiving both i and $i+1$ may generate an alert based on these two updates, but the replicated system will not be able to because neither CE has seen both updates.

Scenario			**Ord.**	**Comp.**	**Cons.**
Lossless			$\surd$	$\surd$	$\surd$
Lossy	Non-his.		X	$\surd$	$\surd$
	His.	Cons.	X	X	$\surd$
		Aggr.	X	X	X

Table 1: Single-variable systems under Algo. AD-1.

Finally, the consistency property is violated by the last configuration. Intuitively, aggressive triggering implies that the CEs will substitute missed data updates with older received values in evaluating conditions. Consequently, “extraneous” alerts are generated which would not have been possible if updates were not missed.

4 Enforcing properties

So far, we have used a particular algorithm for the Alert Displayer, namely, Algorithm AD-1. Next we show that, by using alternative AD algorithms, we can alter the system properties.

4.1 Domination

In Section 3.1 we defined three desirable properties for a replicated system. However, even if two systems satisfy the same set of properties (e.g., both are ordered, consistent, but not complete), one can sometimes be considered “better” than the other. For example, it is easy to devise an AD algorithm to guarantee orderedness and consistency for any system: the AD algorithm will simply not pass any alert through. Since the empty sequence is ordered and is a subsequence of any other sequence, the resulting system is trivially ordered and consistent. Yet if another system also guarantees these properties, while able to generate some alerts to the user, it will be more useful than the one that passes no alert through.

To capture this concept of “goodness” based on how few alerts are filtered out by the AD, we define a relationship **dominates** (denoted by $\geq$) between two AD filtering algorithms. An algorithm $G1$ dominates $G2$ if, given the same input into the AD (multiple interleaved alert sequences), $G1$ always produces a supersequence of $G2$'s output. In short, a dominant algorithm filters fewer alerts and lets more pass through to the final A. We also say that $G1$ **strictly dominates** $G2$ ($G1 > G2$) if $G1 \geq G2$ and for some inputs, $G1$ produces a strict supersequence of $G2$'s output. Therefore, all else being the same (i.e., $G1$ and $G2$ having the same properties), if $G1 > G2$, $G1$ is considered a "better" algorithm than $G2$.

Sometimes, an algorithm is used to improve the properties of a system (e.g., Algorithm AD-2 below enforces orderedness). When a property, such as orderedness, is gained, usually a tradeoff is involved because the AD filters more alerts, and thus the system displays fewer alerts to the user. Therefore, the concept of domination will also be used in the following subsections to show such tradeoffs between different AD algorithms.

4.2 Guaranteeing orderedness

In this subsection, we develop a new AD filtering algorithm, AD-2, which guarantees orderedness in all systems, regardless of whether the front links are lossy or lossless, or whether the condition is conservative or aggressive, etc. Algorithm AD-2 (Figure A-2 in Appendix A) discards any alert that arrives at the AD out of order. Trivially, AD-2 always guarantees that A is ordered.

There may be many AD algorithms that will enforce orderedness. Ideally, we would like to find one that not only guarantees orderedness, but also discards as few alerts as possible. More rigorously, an AD algorithm G is **maximally ordered** if

1. G is ordered, which means that any system using algorithm G is always ordered;
2. There does not exist another algorithm G' such that G' is ordered and $G' > G$.

Theorem 5 *Algorithm AD-2 is maximally ordered.*

Theorem 5 tells us that, if we are looking for an algorithm to guarantee orderedness under all circumstances, we are not going to find any other that is strictly "better" than AD-2.

Table 2 shows the updated system properties under Algorithm AD-2. Comparing with Table 1, we see that a tradeoff is involved. Algorithm AD-2 enforces orderedness at the cost of possibly dropping more alerts (those that arrive out of order) than Algorithm AD-1. This tradeoff is more formally captured in Theorem 6 below.

Scenario			**Ord.**	**Comp.**	**Cons.**
Lossless			$\surd$	$\surd$	$\surd$
Lossy	Non-his.		$\surd$	X	$\surd$
	His.	Cons.	$\surd$	X	$\surd$
		Aggr.	$\surd$	X	X

Table 2: Single-variable systems under Algo. AD-2.

Theorem 6 *AD-1 $>$ AD-2*

In particular, a system with non-historical conditions is no longer complete, as can be illustrated by the following example.

Example 2 Suppose the condition is $c1$: "reactor temperature is over 3000 degrees," and $U_1 = \langle 1^x(3100) \rangle$ and $U_2 = \langle 2^x(3200) \rangle$. Thus $A_1 = T(U_1) = \langle a_1 \rangle$ and $A_2 = T(U_2) = \langle a_2 \rangle$. Assume alert a_2 arrives at the AD before a_1. According to Algorithm AD-2, a_1 will be filtered out, so A is only $\langle a_2 \rangle$. Hence the system is incomplete because $T(U_1 \sqcup U_2) = T(\langle 1^x, 2^x \rangle) = \langle a_1, a_2 \rangle$ has two alerts. ∎

Instead of discarding alerts that arrive out of order (which results in fewer alerts being displayed), the AD could choose to hold off displaying an alert until all its predecessors have been received first. In the previous example, the AD could postpone displaying alert a_2 until it has received a_1, at which time it will display both alerts in order. However, the problem with this approach is that in normal situations the AD has no way of knowing which alerts there are (the alert sequence numbers are not consecutive), causing indefinite delays in some cases. As an improvement, the AD could preset a timeout value t: at most t time after it receives an alert a, it must display a even though a's predecessors might not have all been received. The problem then is, of course, that unless system delays are bounded, orderedness is no longer guaranteed when the AD is forced to display an alert on timeout. Therefore, we conclude that such "delayed displaying" alternatives do not provide anything fundamentally new to our framework, and thus will be left out of our discussions.

4.3 Guaranteeing consistency

Next we describe an AD algorithm, AD-3, that guarantees consistency (Figure A-3). Intuitively, the AD tries to avoid displaying two alerts that require a certain update to be in a "conflicting state." We use an example to illustrate how Algorithm AD-3 works.

Example 3 Suppose that the AD receives from CE1 an alert a_1 with $a_1.H = \langle 3^x, 1^x \rangle$. That is, a_1 triggered on updates 1^x and 3^x, while 2^x was missed by CE1. The AD passes a_1 on to the user. At the same time, it records the situation under which a_1 was generated, by inserting 1 and 3 into its `Received` set, as well as 2 into its `Missed` set.

Later another alert a_2 arrives from CE2 with $a_2.H = \langle 3^x, 2^x \rangle$. If the AD were to pass a_2 through, it would need to put 2 into the `Received` set. However, since 2 is already in `Missed`, this constitutes a "conflicting" state. Thus alert a_2 is filtered out by the AD, as displaying it would lead to an inconsistent output sequence. ∎

We define a **maximally consistent** algorithm analogously to "maximally ordered."

Theorem 7 *Algo. AD-3 is maximally consistent.*

Theorem 8 captures the tradeoff between Algorithms AD-3 and AD-1: AD-3 guarantees consistency while AD-1 can display more alerts to the user.

Theorem 8 *AD-1 > AD-3*

System properties under AD-3 are very similar to Table 1 except that the last row (Aggressive Triggering) is also consistent.

4.4 Combining orderedness and consistency

Finally we combine Algorithms AD-2 and AD-3 to produce a new Algorithm AD-4 (Figure A-4) that guarantees both orderedness and consistency in all scenarios. Simply, AD-4 discards any alert that would be discarded by either AD-2 or AD-3 alone.

Theorem 9 *Algorithm AD-4 is maximally "ordered and consistent."*

System properties under AD-4 are very similar to Table 2 except that Aggressive Triggering also becomes consistent.

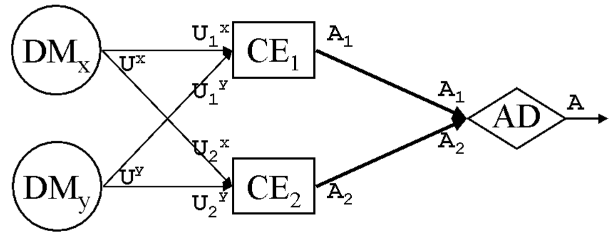


Figure 3: A replicated system monitoring a condition with two variables: x and y.

5 Multi-variable conditions

So far we have dealt with conditions involving only a single variable. Condition expressions containing more than one variables introduce additional complications due to different interleaving of data updates seen by the replicated CEs. Figure 3 shows a system with two independent data sources, x and y.

Appendix C extends the definitions of orderedness, completeness and consistency to multi-variable conditions. Based on those definitions, we have,

Theorem 10 *A multi-variable replicated monitoring system using Algorithm AD-1 is neither ordered nor consistent (hence not complete either).*

5.1 Guaranteeing orderedness

Similar to Section 4.2, we develop Algorithm AD-5 (Figure A-5) to guarantee orderedness in a multi-variable system. Algorithm AD-5 can be regarded as a multi-variable version of Algorithm AD-2. For every alert presented to the user, the AD will record its sequence numbers with respect to both x and y. When a new alert comes in, its sequence numbers are checked against these two numbers for any inversion of order in either x or y. If so, the new alert is discarded because displaying it would result in an unordered output. We prove that the algorithm works in Lemma 4 (Appendix B).

Table 3 summarizes the system properties under Algorithm AD-5. Interestingly, not only can AD-5 provide orderedness in all scenarios, it also makes most systems consistent, except systems with an aggressively triggered historical condition. Lemma 5 in Appendix B gives the proof. Additionally, Lemma 6 in Appendix B proves that multi-variable systems under Algorithm AD-5 are still all incomplete.

Scenario			Ord.	Comp.	Cons.
Lossless			$\surd$	X	$\surd$
Lossy	Non-his.		$\surd$	X	$\surd$
	His.	Cons.	$\surd$	X	$\surd$
		Aggr.	$\surd$	X	X

Table 3: Multi-variable systems under Algo. AD-5.

5.2 Orderedness and consistency

Algorithm AD-6 (Figure A-6) enforces both orderedness and consistency in a multi-variable system. Thus its system properties are the same as Table 3 except that the last row (Aggressive Triggering) is also consistent. Like AD-4 for single-variable systems, Algorithm AD-6 combines AD-5, which enforces orderedness, and the multi-variable version of AD-3, which enforces consistency. The extension of Algorithm AD-3 to the multi-variable case is rather straightforward, and is left to the appendix.

6 Related work

Much work has been done on maintaining consistency in the face of replication, at various system levels [9, 7, 3]. Although we deal with a fundamentally different type of replication, our notion of correctness is analogous to the concept of single-copy serializability in distributed database literature [3].

Mechanisms have been proposed in [1, 5] for scalable and reliable content-based publish/subscribe systems. Although our condition monitoring systems are functionally similar, our work concentrates on the impact of replicated independent triggering servers.

Distributed sensor networks [6] monitor changes in an environment. Our work is not concerned with how changes are detected or how the sensors are coordinated. We are concerned with consistency issues with alerts that are derived from such monitoring. It is also possible to modify our algorithms for special cases, such as when the alerted actions are idempotent, as identified by the process-control work [10].

The idea of replicating a service to increase its fault-tolerance in a general purpose system is certainly not new [2, 4, 8]. However, the contribution of this paper is to identify and systematically study the implication of replication in a special type of systems, i.e., distributed condition monitoring systems.

7 Conclusion

In this paper we have studied problems resulting from replication in a condition monitoring system. Replication is used to increase the system's fault tolerance. However, replication can also lead to confusing and conflicting alerts. We define orderedness, consistency, and completeness as three desirable properties of a replicated system. We then study the scenarios under which these properties are maintained. We also propose new algorithms to enhance or tradeoff these properties in various scenarios.

References

[1] M. K. Aguilera, R. E. Strom, D. C. Sturman, M. Astley, and T. D. Chandra. Matching events in a content-based subscription system. In *Proceedings of the 18th Annual ACM Symposium on Principles of Distributed Computing*, pages 53–61, 1999.

[2] J. Bartlett. A NonStop kernel. *Operating Systems Review*, 11.5:23–31, 1977.

[3] P. A. Bernstein and E. Newcomer. *Principles of Transaction Processing*. Morgan Kaufmann, 1997.

[4] A. Borg, W. Blau, W. Graetsch, F. Herrmann, and W. Oberle. Fault tolerance under UNIX. *ACM Transactions on Computer Systems*, 7.1:1–24, 1989.

[5] A. Carzaniga, D. S. Rosenblum, and A. L. Wolf. Achieving scalability and expressiveness in an Internet-scale event notification service. In *Proceedings of the 19th Annual ACM Symposium on Principles of Distributed Computing*, pages 219–27, 2000.

[6] D. Estrin, R. Govindan, J. Heidemann, and S. Kumar. Next century challenges: scalable coordination in sensor networks. In *Proceedings of the Fifth Annual ACM/IEEE International Conf. on Mobile Computing and Networking*, pages 263–270, 1999.

[7] J. Gray, P. Helland, P. O'Neil, and D. Shasha. The dangers of replication and a solution. In *ACM SIGMOD Conference*, pages 173–182, 1996.

[8] Microsoft Cluster Server. `http://www.microsoft.com/ntserver/ntserverenterprise/exec/overview`.

[9] M. Nelson, B. Welch, and J. Ousterhout. Caching in the Sprite network file system. *ACM Transactions on Computer Systems*, 6.1:134–154, 1988.

[10] M. Polke, editor. *Process control eng.* VCH, 1994.

Algorithm AD-1 (Exact Duplicate Removal)

```
P = {}    // the empty set
On receiving new alert a:
  if a is in P
    discard a
  else
    P = P + {a}
    add a to output sequence A
```

Figure A-1: Algorithm AD-1

Algorithm AD-2

```
last = -1
On receiving new alert a:
  if a.seqno.x <= last
    discard a
  else
    last = a.seqno.x
    add a to output sequence A
```

Figure A-2: Algorithm AD-2

Appendix A AD algorithm details

In this section, we give details (pseudo-code) for the AD algorithms proposed in this paper. Algorithm AD-1 (Figure A-1) only eliminates exact duplicates. Algorithms AD-2, AD-3 and AD-4 (Figures A-2, A-3 and A-4) guarantee orderedness, consistency, and both in a single variable system, respectively. Algorithm AD-5 (Figure A-5) enforces orderedness in a multi-variable system, while AD-6 (Figure A-6) enforces both orderedness and consistency. Note that the pseudo-code for AD-5 assumes a two-variable condition, although the algorithm can be easily extended for conditions with more than two variables.

Appendix B Proofs

In this section, we prove the various lemmas and theorems given earlier in this paper. We use M_{AD-i} to denote the merging and filtering performed by the Alert Displayer under AD Algorithm AD-i. Thus, $A = M_{AD-i}(A_1, A_2)$. If the AD algorithm is implied in the context, we will abbreviate it to simply M. Note that function M is timing-dependent, in the sense that its output A not only depends on the two inputs A_1 and A_2, but also on the way A_1 and A_2 are interleaved. For example, if $A_1 = \langle a_1, a_2 \rangle$ and $A_2 = \langle a_1, a_3 \rangle$, then A can be $\langle a_1, a_2, a_3 \rangle$ or $\langle a_1, a_3, a_2 \rangle$. When we prove properties for M below, the result applies to all possible outcomes of M regardless of how its input sequences are interleaved.

Before proving our first theorem, Theorem 1, we introduce a few lemmas. Note that here M stands for M_{AD-1}.

Algorithm AD-3

```
Received = {}
Missed = {}
On receiving new alert a:
  if Conflicts(a.history)
    discard a
  else
    UpdateState(a.history)
    add a to output sequence A
```

where

```
Conflicts(H) {
  foreach sequence number s in Hx
    if (s in Missed)
      return True;

  foreach s in SpanningSet(Hx)
    if (s not in Hx  AND  s in Received)
      return True;

  return False;
}

UpdateState(H) {
  Received = Received + Hx;
  Missed = Missed +
             (SpanningSet(Hx) - Hx);
}
```

`SpanningSet(s)` *is the set of consecutive integers between the smallest and the biggest elements in a set s, inclusive. For example,* `SpanningSet(`$\{1,2,5\}$`)` = $\{1,2,3,4,5\}$.

Figure A-3: Algorithm AD-3

Algorithm AD-4 *Algorithm AD-4 removes any alert that would be removed by either Algorithm AD-2 or AD-3. The pseudo-code is not given here.*

Figure A-4: Algorithm AD-4

Algorithm AD-5

```
lastx = -1
lasty = -1
On receiving new alert a:
  if Conflicts(a)
    discard a
  else
    UpdateState(a)
    add a to output sequence A
```

where

```
Conflicts(a) {
  if (a.seqno.x < lastx OR
      a.seqno.y < lasty)
    return True;    // Conflicting

  if (a.seqno.x == lastx AND
      a.seqno.y == lasty)
    return True;    // Duplicate

  return False;
}

UpdateState(a) {
  lastx = a.seqno.x;
  lasty = a.seqno.y;
}
```

Figure A-5: Algorithm AD-5

Algorithm AD-6 *Algorithm AD-6 combines Algorithm AD-5 with the multi-variable version of Algorithm AD-3. To extend Algorithm AD-3 to multi-variable systems, the AD keeps two lists (*`Received` *and* `Missed`*) each for variable x and variable y. The pseudo-code is not given here.*

Figure A-6: Algorithm AD-6

Lemma 1 $\Phi M(A_1, A_2) = \Phi A_1 \cup \Phi A_2$*, for any* A_1 *and* A_2.

Proof: Part 1: $\Phi M(A_1, A_2) \subset \Phi A_1 \cup \Phi A_2$. Because any alert in $M(A_1, A_2)$ has to have come from either A_1 or A_2.

Part 2: $\Phi A_1 \cup \Phi A_2 \subset \Phi M(A_1, A_2)$. Because AD-1 only removes an alert if an identical one is already in the output. ■

Corollary 1 $M(A, A) = A$*, for any ordered sequence* A.

Proof: Part 1: $\Phi M(A, A) = \Phi A$. By Lemma 1, $\Phi M(A, A) = \Phi A \cup \Phi A = \Phi A$.

Part 2: $M(A, A)$ is ordered. Proof by contradiction. Suppose the first inversion of order in $M(A, A)$ happens at an element whose value is i, followed by another number j, such that $i > j$. We name the two identical input streams A_1 and A_2 ($A_1 = A_2 = A$). Hence both numbers i and j appear in both A_1 and A_2. Without loss of generality, suppose that the i in $M(A, A)$ is a result of the i in A_1. Because A_1 is ordered, the number j must appear before i in A_1. Because $AD - 1$ only removes duplicates, and sends along all numbers that have not been seen, when j from A_1 is encountered by the AD, it must put j in the output sequence (or j must have already been put in the output). This means that j must also appear before i in the output. Since $M(A, A)$ cannot contain duplicates, we have just arrived at a contradiction. ■

Lemma 2 $U \sqcup U = U$*, for ordered sequence* U.

Proof: The lemma follows from the definition of "$\sqcup$". ■

Now we are ready to prove Theorem 1.

Proof (Theorem 1): Part 1: completeness. Because the front links are lossless, we have $U_1 = U_2 = U$. So, $U_1 \sqcup U_2 = U \sqcup U = U$ according to Lemma 2 above. We also have $A_1 = T(U_1) = T(U)$ and $A_2 = T(U_2) = T(U)$. Finally, by Corollary 1, $A = M(A_1, A_2) = M(T(U), T(U)) = T(U) = T(U_1 \sqcup U_2)$. It follows that $\Phi A = \Phi T(U_1 \sqcup U_2)$.

Part 2: orderedness. A is ordered because $A = T(U)$ and U is ordered. ■

The following lemma is needed by the proof of Theorem 2.

Lemma 3 *If a mapping* T *is non-historical,* $T(U_1 \sqcup U_2) = T(U_1) \sqcup T(U_2)$.

Proof: Part 1: $\Phi T(U_1 \sqcup U_2) = \Phi(T(U_1) \sqcup T(U_2))$. $\Phi T(U_1 \sqcup U_2) \subset \Phi(T(U_1) \sqcup T(U_2))$ because an alert a in $T(U_1 \sqcup U_2)$ must be triggered by an update in either U_1

or U_2 (since T is non-historical). Hence a must belong to either $T(U_1)$ or $T(U_2)$. On the other hand, $\Phi(T(U_1) \sqcup T(U_2)) \subset \Phi T(U_1 \sqcup U_2)$ because if an alert is in $T(U_1)$, it must also be in $T(U_1 \sqcup U_2)$. Analogously for $T(U_2)$.

Part 2: both $T(U_1 \sqcup U_2)$ and $T(U_1) \sqcup T(U_2)$ are ordered. Because both U_1 and U_2 are ordered, $U_1 \sqcup U_2$ is also ordered. Furthermore, T of an ordered sequence is an ordered sequence (alerts are given out in order). ∎

Corollary 2 *For a non-historical T, we have: $\Phi T(U_1 \sqcup U_2) = \Phi T(U_1) \cup \Phi T(U_2)$.*

Proof: Since $T(U_1 \sqcup U_2) = T(U_1) \sqcup T(U_2)$ by lemma above, we have $\Phi T(U_1 \sqcup U_2) = \Phi(T(U_1) \sqcup T(U_2)) = \Phi T(U_1) \cup \Phi T(U_2)$. ∎

Proof (Theorem 2): Part 1: completeness.

$$\begin{aligned} \Phi A &= \Phi M(A_1, A_2) \\ &= \Phi A_1 \cup \Phi A_2 (\text{Lemma 1}) \\ &= \Phi T(U_1) \cup \Phi T(U_2) \\ &= \Phi T(U_1 \sqcup U_2) (\text{Corollary 2}) \end{aligned}$$

Part 2: unorderedness. Proof with a counter-example. Assume condition $c1$ ("reactor temperature is over 3000 degrees"). Suppose $U = \langle 1(3100), 2(3500) \rangle$. CE1 receives both updates, whereas update 1 is missed by CE2. In other words, $U_1 = U$ but $U_2 = \langle 2 \rangle$. Consequently, $A_1 = T(U_1) = \langle 1, 2 \rangle$, and $A_2 = T(U_2) = \langle 2 \rangle$. If alert 2 from A_2 arrives at the AD before both alerts in A_1, $A = M(A_1, A_2) = \langle 2, 1 \rangle$. Obviously this A is an unordered sequence. ∎

Proof (Theorem 3): Part 1: consistency. We show that by choosing $U' = U_1 \sqcup U_2$, we will always have $\Phi A \subseteq \Phi T(U')$. For any alert $a \in A$, we must have $a \in A_1$ or $a \in A_2$ (or both). Without loss of generality, suppose that $a \in A_1$. It follows that U_1 contains a series of updates $i-(d-1), i-(d-2), \ldots, i$ which triggered a, where d is the degree of the condition. Note that the updates have to be consecutive because the condition is conservative.

Because $\langle i-(d-1), i-(d-2), \ldots, i \rangle \sqsubseteq U_1$, we also get $\langle i-(d-1), i-(d-2), \ldots, i \rangle \sqsubseteq U_1 \sqcup U_2$. Hence $a \in T(U_1 \sqcup U_2) = T(U')$. Consequently, $\Phi A \subseteq \Phi T(U')$.

Part 2: incompleteness and unorderedness. The following example shows that the system is neither complete nor ordered. Assume condition $c3$ ("reactor temperature has risen for more than 200 degrees since last reading taken at the DM"). Let $U_1 = \langle 1(1000), 2(1500) \rangle$ and $U_2 = \langle 3(2000), 4(2500) \rangle$. It follows that $A_1 = T(U_1) = \langle 2 \rangle$, and $A_2 = T(U_2) = \langle 4 \rangle$. They are merged to produce A, which is $\langle 4, 2 \rangle$ or $\langle 2, 4 \rangle$. On the other hand, $T(U_1 \sqcup U_2) = T(\langle 1, 2, 3, 4 \rangle) = \langle 2, 3, 4 \rangle$. Thus the system violates both completeness and orderedness. ∎

Proof (Theorem 4): Violation of orderedness can be shown in a similar example as in Theorem 3. Here we give a counter-example illustrating the lack of consistency.

Assume condition $c2$ ("reactor temperature has risen for more than 200 degrees since last reading received"). Let $U = \langle 1(400), 2(700), 3(720) \rangle$, and $U_1 = U$, but $U_2 = \langle 1, 3 \rangle$. Consequently, $A_1 = T(U_1) = \langle 2 \rangle$, and $A_2 = T(U_2) = \langle 3 \rangle$. Algorithm AD-1 will produce A as either $\langle 2, 3 \rangle$ or $\langle 3, 2 \rangle$. We now show that no U' exists such that $\Phi A = \{2, 3\} \subseteq \Phi T(U')$. In particular, the presence of alert 2 in $T(U')$ implies that update 2 must belong in U'. However, alert 3 means that update 2 must not be in U', a contradiction. ∎

Proof (Theorem 5): Part 1: AD-2 is ordered. It follows from the construction of the algorithm.

Part 2: no G' exists such that $G' >$ AD-2. Proof by contradiction. Suppose such a G' exists. Then there must exist one input sequence which causes G' to produce a strict supersequence (A') of AD-2's output (A). Let a' be the first alert that satisfies $a' \in A'$ but $a' \notin A$. Because AD-2 filters out a', we must have $a'.seqno.x < a_p.seqno.x$ where a_p is the alert preceding a' in A. Since $a_p \in A'$ (because $A' \sqsupseteq A$), A' must not be ordered, a contradiction. ∎

Proof (Theorem 6): AD-1 $\geq$ AD-2 because any alert filtered out by Algorithm AD-1 will also be filtered out by AD-2. Recall that AD-1 simply removes duplicates while AD-2 removes both duplicates and out-of-order alerts.

Finally, AD-1 $>$ AD-2 because for some inputs, namely out-of-order sequences, AD-2 removes strictly more alerts than AD-1. ∎

Proof (Theorem 7): Part 1: AD-3 is consistent. Let A denote an alert sequence produced by a replicated one-variable system with Algorithm AD-3. We now prove that the system is consistent by showing that $\Phi A \subseteq \Phi T(R)$, where R is the `Received` set recorded at the AD after A is produced. In other words, we have chosen the U' in the definition of consistency (Section 3.1) to be R. Obviously $R \sqsubseteq (U_1 \sqcup U_2)$.

For each alert $a \in A$, we will show that $a \in T(R)$ as well. Without loss of generality, assume that the condition in question is of degree 2. Suppose a triggered on two updates i and j. Since Algorithm AD-3 did not filter out a, we must have $i, j \in R$ and $k \notin R$ for all $i < k < j$. Consequently, a must be in $T(R)$.

Part 2: no G' exists such that $G' >$ AD-3. Proof by contradiction. Assume such a G' exists, and we will show that G' cannot be consistent. There exists one input sequence which causes G' to produce a strict supersequence (A') of AD-3's output (A). Let a' be the first alert that satisfies $a' \in A'$ but $a' \notin A$. Further assume that a triggered on alerts i and j with $i < j$. Because a' was filtered out by Algorithm AD-3, one of the following cases must hold:

CASE 1: $i \in M$, where M is the `Missed` set in the algorithm. Let a_p denote the alert in A which caused i to be placed in `Missed`. Since $A' \sqsupseteq A$, $a_p \in A'$. We argue that any system that produces an output containing both a' and a_p cannot be consistent. Specifically, if U' contains i, a_p cannot be in $T(U')$. Similarly, if $i \notin U'$, then $a' \notin T(U')$.

CASE 2: $j \in M$. Analogous to Case 1.

CASE 3: $\exists k$ such that $i < k < j$ and $k \in R$. Similar to the previous two cases, there must exist an alert a_p which depends on k to have been received. This creates a conflict which makes G' inconsistent. ■

Proof (Theorem 8): Analogous to Theorem 6, we observe that AD-3 filters out at least all the alerts filtered by AD-1, and more in some cases. ■

Proof (Theorem 9): The maximality proof for Algorithm AD-4 parallels that for its two parent algorithms: AD-2 and AD-3. We omit the proof details due to space limitations. ■

Proof (Theorem 10): We prove by giving a concrete counter-example. A two-variable system (Figure 3) has two DMs x and y, representing two reactors' temperature sensors. Suppose that over a period of time, $U^x = \langle 1^x(1000), 2^x(1200)\rangle$, and $U^y = \langle 1^y(1050), 2^y(1150)\rangle$. Assume that no updates are lost. That is, $U_1^x = U_2^x = U^x$ and $U_1^y = U_2^y = U^y$. Further assume that streams U_1^x and U_1^y are interleaved at CE1 such that $U_1 = \langle 1^x, 2^x, 1^y, 2^y\rangle$. However, a different interleaving at CE2 makes $U_2 = \langle 1^y, 2^y, 1^x, 2^x\rangle$. The different interleaving could be due to network delays between certain data sources and certain CEs.

The condition triggers whenever the temperature difference between the two reactors exceeds 100 degrees. More formally, $c_m(H) = (\mid H_x[0].value - H_y[0].value \mid > 100)$. Notice that the condition is of degree one to variable x and to variable y. We have $A_1 = \langle a(2^x, 1^y)\rangle$, and $A_2 = \langle a(1^x, 2^y)\rangle$, where $a(i^x, j^y)$ stands for an alert that triggers on x-update i and y-update j. When A_1 and A_2 are merged by Algorithm AD-1, both alerts are presented to the user. Without loss of generality, assume that $A = \langle a(2^x, 1^y), a(1^x, 2^y)\rangle$.

We first note that such a system is unordered because A is not ordered with respect to x ($\Pi_x A = \langle 2, 1\rangle$ is unordered). The system is also inconsistent because no possible non-replicated system could generate such two alerts. If alert $a(2^x, 1^y)$ appears before $a(1^x, 2^y)$, update 2^x must be received at the CE before 1^x, which is not possible. Likewise, $a(1^x, 2^y)$ before $a(2^x, 1^y)$ would require update 2^y to be received before 1^y. ■

Lemma 4 *Algorithm AD-5 results in an ordered two-variable system.*

Proof: If a_1 and a_2 are two consecutive alerts in the final output sequence A, then we have $a_1.seqno.x <= a_2.seqno.x$ because otherwise a_2 would have been filtered out by Algorithm AD-5. Hence A is ordered with respect to x. Analogously for y. ■

Lemma 5 *A system with Algorithm AD-5 is consistent unless the condition is historical and aggressively triggered.*

Proof: Part 1: consistency. Due to space constraint, we only give the proof for systems with lossless links. Proofs for other types of systems can be similarly constructed.

Let A denote an output produced by such a replicated system given x-updates U_x and y-update U_y. We will show that it is possible to construct an interleaving of U_x and U_y, called U_V, such that $\Phi A \subseteq \Phi T(U_V)$. Suppose $A = \langle a_1, a_2, \ldots, a_p\rangle$. Let x_i be a shorthand for $a_i.seqno.x$, and y_i for $a_i.seqno.y$. We observe that alert $a_i \in T(U_V)$ if and only if $x_i^x \rightarrow (y_i+1)^y$ and $y_i^y \rightarrow (x_i+1)^x$ in U_V, where $\rightarrow$ denotes precedence in the interleaving. As an example, if alert a_4 triggered on updates 4^x and 8^y, then $T(U_V)$ will contain a_4 if and only if 4^x comes before 9^y and 8^y comes before 5^x in U_V.

Consequently, $\Phi A \subseteq \Phi T(U_V)$ requires that

$$\forall i, 1 \le i \le p, x_i^x \rightarrow (y_i+1)^y \text{and} y_i^y \rightarrow (x_i+1)^x$$

(Requirement 1). Additionally, since U_V must be ordered, we have

$$\forall j, j^x \rightarrow (j+1)^x \text{and} j^y \rightarrow (j+1)^y$$

(Requirement 2). As long as we show that an U_V exists which satisfies the above two requirements, we have proven that the system is consistent.

We observe that the only possibility that such an U_V cannot be found is if the two requirements create a loop in the precedence graph of all the updates. We claim that the loop must be of the form:

$$m^x \rightarrow (m+s)^x \rightarrow n^y \rightarrow (n+t)^y \rightarrow m^x \text{where} s, t \ge 0$$

Because $(m+s)^x \rightarrow n^y$, A must contain an alert a_u which triggered on x-update $m+s$ and y-update $n-1$. Similarly, $(n+t)^y \rightarrow m^x$ implies that another alert a_v in A triggered on $(m-1)^x$ and $(n+t)^y$. Since $m+s > m-1$ but $n-1 < n+t$, the resulting A cannot be ordered no matter what the relative order of a_u and a_v is in A. This is contrary to Theorem 4. Hence such a loop in the precedence graph cannot exist and we have proven our proposition.

Part 2: inconsistency when condition is aggressive. A counter-example similar to that in the proof of Theorem 4 can be constructed. The reader is referred to that proof for details. ■

Lemma 6 *A system with Algorithm AD-5 is incomplete.*

Proof: We prove by giving a counter-example. We assume a non-historical condition and lossless front links for simplicity. Examples for other types of systems can be similarly constructed.

Assume that the condition is satisfied by only three pairs of updates: $(8^x, 2^y)$, $(8^x, 3^y)$ and $(8^x, 4^y)$. CE1 sees the following update sequence: $\langle 8^x, 2^y, 9^x, 3^y, 4^y, \dots \rangle$, and generates one alert: $a(8^x, 2^y)$. However, CE2 sees a different interleaving of the updates: $\langle 2^y, 3^y, 7^x, 4^y, 8^x, \dots \rangle$ and generates a different alert: $a(8^x, 4^y)$.

Next we show that an interleaving of the updates (U_V) cannot be found such that $\Phi T(U_V)$ contains only $a(8^x, 2^y)$ and $a(8^x, 4^y)$, but not $a(8^x, 3^y)$. By the same notation as in Lemma 5, $a(8^x, 2^y) \in T(U_V)$ implies that $8^x \rightarrow 3^y$ in U_V. Since $3^y \rightarrow 4^y$, we get $8^x \rightarrow 4^y$. Similarly, $(8^x, 4^y) \in T(U_V)$ means that $3^y \rightarrow 4^y \rightarrow 9^x$. Consequently, $(8^x, 3^y)$ must also belong in $T(U_V)$. Therefore the required U_V cannot be found and the system is not complete. ∎

We omit the correctness proof for Algorithm AD-6 to avoid redundancy.

Appendix C Multi-variable definitions

We give the multi-variable definitions of the three desirable system properties. Note that these are direct extensions of the single-variable case, and fall back to the single-variable definitions when the number of variables is one. A replicated multi-variable system R is said to have each of the following properties if every alert sequence A it produces satisfies the corresponding criterion.

1. **Orderedness:** A is ordered (with respect to every variable in V).
2. **Completeness:** $\Phi A = \Phi T(U_V)$, where U_V is any interleaving of several data update sequences, one for each variable in V. The update sequence for variable x is the ordered union of x-updates received by all the CEs in the system.
3. **Consistency:** $\exists U'$ such that $\Phi A \subseteq \Phi T(U')$ and $U' \sqsubseteq U_V$ where U_V is as defined above.

Appendix D Multiple conditions

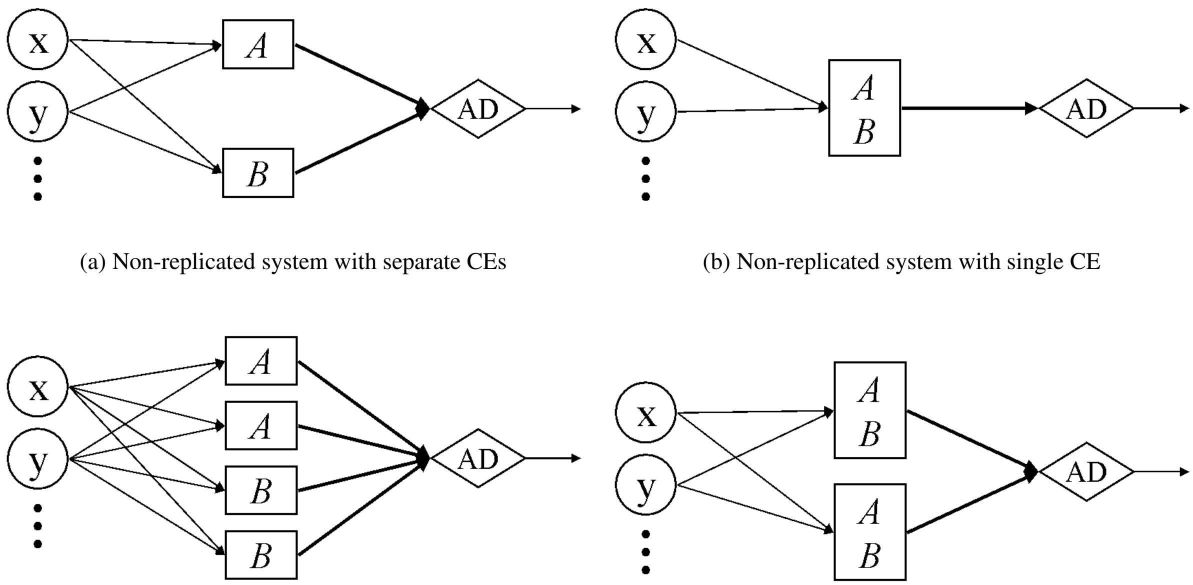
When multiple conditions are monitored in a system, additional complications arise. A thorough study of multiple condition systems is beyond the scope of this paper. In this appendix we merely motivate further study by pointing out some of the problems. The following example illustrates that, even without replication, a multi-condition system can have issues of consistency.

Example 4 Figure D-7(a) illustrates a system with two conditions A and B. Suppose A is the condition "reactor x has a higher temperature than reactor y," and B represents "y has a higher temperature than x." Initially, the two reactors have the same temperature of 2000 degrees. Later both raise to 2100 degrees. However, the CE for condition A sees the x temperature change first, and triggers. In contrast, the CE for B sees the y change first, and also triggers. If both alerts are reported to the user, the user will be confused by the conflicting messages. This is an example where each condition alone triggers in a sensible way, but the combination of the two paints a conflicting picture to the user because the conditions are interdependent. ∎

When replication is added to a multi-condition system, more issues surface. To begin with, a "corresponding non-replicated system" is no longer well defined. For example, assume that two conditions, A and B, are being monitored. Both systems in Figures D-7(a) and D-7(b) are non-replicated systems that monitor both conditions. In Figure D-7(a), the CEs for A and for B reside on separate nodes (thereby allowing them to see different sets of updates), while Figure D-7(b) has the two CEs co-located on the same node. Consequently, when we extend our analysis to multi-condition replicated systems, it is not apparent which one we should choose as the base for such definitions as consistency and completeness.

We observe that many of the issues can be solved using similar approaches as those proposed in this paper. For example, we can apply our single-condition analysis and algorithms to Figure D-7(c) if we use Figure D-7(a) as its corresponding non-replicated system. The reason is that, in both Figures D-7(c) and D-7(a), the two conditions A and B can be thought of as totally separate from each other. Specifically, each CE behaves exactly like in the single-condition case, because it only needs to monitor one condition (either A or B). Although there is only one AD for both conditions, it can effectively separate the A and B alert streams and run one instance of the filtering algorithm against each stream.

Analogously, Figure D-7(d) can be dealt with using Figure D-7(b) as its corresponding non-replicated system. Informally, we devise a new condition $C = A \vee B$. That is, condition C triggers whenever A triggers or B triggers. Effectively, systems D-7(d) and D-7(b) have been reduced to D-8(b) and D-8(a), respectively, which are single condition systems.



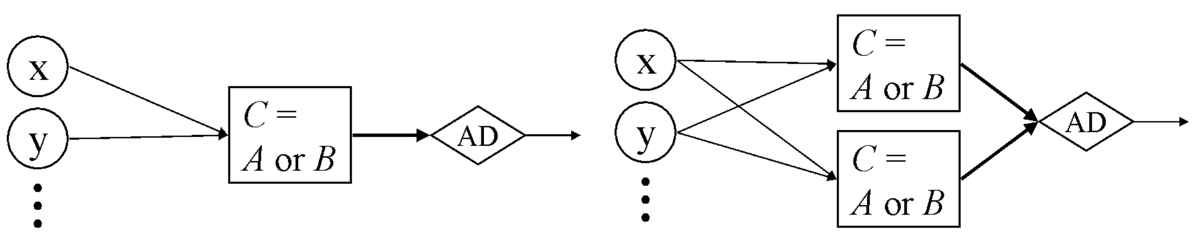
(a) Non-replicated system with separate CEs

(b) Non-replicated system with single CE

(c) Replicated system with separate CEs

(d) Replicated system with co-located CEs

Figure D-7: Systems that monitor two conditions: A and B.



(a) Same as Figure D-7(b)

(b) Same as Figure D-7(d)

Figure D-8: Two co-located conditions A and B can be regarded as one combined condition $C = A \vee B$.