Decentralized Computation of Threshold Crossing Alerts

Fetahi Wuhib[1], Mads Dam[1], Rolf Stadler[1], and Alexander Clemm[2]

[1] KTH Royal Institute of Technology,
Stockholm, Sweden
{fzwuhib,mfd,stadler}@kth.se
[2] Cisco Systems
San Jose, California, USA
alex@cisco.com

Abstract. Threshold crossing alerts (TCAs) indicate to a management system that a management variable, associated with the state, performance or health of the network, has crossed a certain threshold. The timely detection of TCAs is essential to proactive management. This paper focuses on detecting TCAs for network-level variables, which are computed from device-level variables using aggregation functions, such as SUM, MAX, or AVERAGE. It introduces TCA-GAP, a novel protocol for producing network-wide TCAs in a scalable and robust manner. The protocol maintains a spanning tree and uses local thresholds, which adapt to changes in network state and topology, by allowing nodes to trade unused "threshold space". Scalability is achieved through computing the thresholds locally and through distributing the aggregation process across all nodes. Fault-tolerance is achieved by a mechanism that reconstructs the spanning tree after node addition, removal or failure. Simulation results on an ISP topology show that the protocol successfully concentrates traffic overhead to periods where the aggregate is close to the given threshold.

1 Introduction

Threshold crossing alerts (TCAs) indicate to a management system that some monitored MIB object, or management variable, has crossed a certain preconfigured value - the threshold. Objects that are monitored for TCAs typically contain performance-related data, such as link utilization or packet drop rates. In order to avoid repeated TCAs in case the monitored variable oscillates, a threshold is typically accompanied by a second threshold called the hysteresis threshold, set to a lower value than the threshold itself. The hysteresis threshold must be crossed, in order to clear the TCA and allow a new TCA to be triggered when the threshold is crossed again (Fig. 1).

TCAs represent an important mechanism in proactive management, as they allow for management that is event-based and does not need to rely as much on centralized polling.

Today, TCAs are generally set up per device, e.g., for monitoring packet drop rates on a particular link. In addition, Service Level Agreements (SLAs) are often articulated similarly, on a per-device basis, reflecting the limitations of today's technology. However, there is a definitive need for management functionality that provides cross-device TCAs, which are applied to parameters that are aggregated across the network. Examples include management applications that alert an operator whenever (a) the average link utilization in a domain rises above certain threshold, or (b) whenever the number

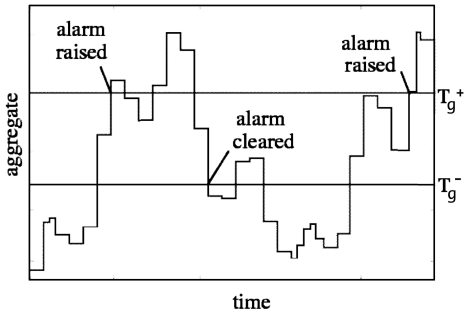


Fig. 1. Threshold Crossing Alerts: an alert is raised when a network-wide variable, the aggregate, exceeds a given global threshold T_g^+. The alert is cleared when the aggregate has decreased below a lower threshold T_g^-.

of currently active voice calls on a network, as aggregated across IP PBXs or voice gateways, exceeds a given value.

This work focuses on supporting TCAs for thresholds on network-wide management variables, which are computed by aggregating local variables across many devices. We will refer to such TCAs as *network TCAs (NTCAs)* and to network-wide management variables as *aggregates*. Typical NTCAs involve aggregates that are computed from device variables, using functions, such as SUM, AVERAGE, COUNT, MAX, or MIN. (For a discussion on the practical relevance of NTCAs, see [1].)

The hard part in determining when to trigger NTCAs is to ensure scalability and fault-tolerance of the approach. Traditionally, the aggregation of local variables from different devices has been performed in a centralized way, whereby an application, running on a management station, first retrieves state variables from agents in network devices and then aggregates them on the management station. Such an approach has well-known drawbacks with respect to scalability and fault tolerance.

We propose that NTCAs be computed in a distributed way. To this end, we assume that each network device participates in the computation by running a management process, either internally or on an external, associated device. These management processes communicate via an overlay network for the purpose of monitoring the network threshold. Throughout the paper, we refer to this overlay as the *network graph*. A node in this graph represents a management process together with its associated network device(s). While the topology of this overlay can be chosen independently from the topology of the underlying physical network, we assume in this paper, for simplicity, that both topologies are the same, i.e., that the management overlay has the same topology as the as the underlying physical network.

A straightforward solution to the NTCA problem can be constructed by using a protocol for distributed state aggregation, such as [2, 3]. These protocols provide a continuous estimate of the network-wide aggregate on a dedicated root node, by setting up a spanning tree on the network graph, along which updates are reported. NTCAs can be detected by a filter on the root node. However, such a solution is inherently inefficient in terms of traffic overhead on the network graph and processing load on the management nodes. For the purpose of triggering NTCAs, we are not interested in receiving estimates about the dynamically changing aggregate, but only in receiving alarms when it crosses certain thresholds. For instance, no estimate of the aggregate is needed if its value is well below a threshold.

In this paper, we present TCA-GAP, a novel protocol for computing NTCAs in a scalable and robust manner. The protocol is based on the Generic Aggregation Protocol (GAP), which provides support for creating and maintaining a spanning tree on the network graph and for distributed aggregation of local variables [3]. The basic idea behind our protocol is the use of local thresholds that control whether a node reports a change in aggregate of its subtree. These thresholds are locally recomputed whenever local threshold rules are violated, which can be triggered, e.g., by a "significant change" in a device variable or a node failure. Scalability is achieved through computing the thresholds locally and through distributing the aggregation process across all nodes of the spanning tree. Fault-tolerance is achieved by a mechanism that reconstructs the spanning tree after node addition, removal or failure. We evaluate the protocol on an ISP topology and compare its performance to a naïve solution to the NTCA problem and to a centralized scheme for NTCA detection proposed by Dilman and Raz [4].

The paper is organized as follows. Section 2 reviews related work. Section 3 formally defines the NTCA problem. Section 4 provides GAP in a nutshell, and section 5 presents our protocol. Section 6 gives simulation scenarios, simulation results and a discussion of those results. Finally, section 7 provides some additional comments to the results and gives an outlook on further work.

2 Related Work

Dilman and Raz [4] study the NTCA problem for a centralized management system, where the management station communicates directly with the network elements. The authors assume that the aggregation function is sum and that a single global threshold T is given. In one solution, which the authors call 'simple-value', local threshold value of T/n where n is the number of nodes in the network is assigned to all nodes. Whenever the local weight becomes larger than this threshold, the node sends a trap with the current weight to the management station. Periodically, if the management station has received traps during the previous period, it polls all nodes that did not send a trap for their local weights. Then, it aggregates the local weights of all nodes and determines whether the global threshold has been crossed. This scheme performs well for "small" networks, where polling is feasible, where weights are evenly distributed, and where the likelihood of a node exceeding its threshold is small. In 6 we compare the performance of TCA-GAP to this simple-value scheme for a specific scenario. In the same paper, the authors propose a second scheme, called 'simple-rate', which assumes an upper bound on $\Delta w/\Delta t$, the range of change of weights per unit time. This assumption leads to an upper bound on the aggregate and thus allows nodes to be sampled less frequently.

Decentralized solutions for problems closely related to the NTCA problem have been proposed by Breitgand, Dolev and Raz in the context of estimating the size of a multicast group [5]. There, the task is to determine whether the group size is within a prescribed interval for which pricing is constant. Several schemes are proposed, based on the concept that nodes intercept traps generated by their children, in order to suppress false alerts.

Outside the specific domain of TCA generation, the problem of distributed state aggregation has recently received considerable attention (cf. [2, 6–9]). An approach common to several authors (ourselves including) is to reduce traffic load by installing filters at each node of the aggregation tree. Olston et al. [10] propose a scheme whereby filters installed at each node continually shrink, leaving room for a central processor

to reallocate filter space where needed most. This scheme was later refined in [11], by using statistics on the local aggregates held at each node, in order to allow filters to dynamically adjust to the data sources. One drawback of this approach is that it is not *temporally local*, because the criteria for setting the filters depend on the histories of previously sampled values. This makes the approach vulnerable to failures and dynamic changes, since these can affect the shape of the aggregation tree in unpredictable ways. (This approach though is *spatially local*, since each node makes local decisions to set the filters for its children.) By way of comparison, the solution we propose is both spatially and temporally local and applies a rather simple, history-free scheme to transfer threshold space between siblings in an aggregation tree.

3 The NTCA Problem

We are considering a dynamically changing network graph $G(t) = (V(t), E(t))$ in which nodes $i \in V(t)$ and edges/links $e \in E(t) \subseteq V(t) \times V(t)$ may appear and disappear over time. To each node i is associated a *weight*, $w_i(t) \geq 0$. The term weight is used to represent a local state variable or a device counter that is being subjected to threshold monitoring. For the main part of the paper we assume that weights are integer valued quantities, aggregated using SUM. In section 7 we discuss extensions to support other aggregates such as those mentioned in section 1.

The objective is to raise an alert on a distinguished root node, the management station, when the aggregate weight $\Sigma_i w_i(t)$ exceeds a given global threshold T_g^+, and to clear the alert on the root when the aggregate has decreased below a lower threshold T_g^-.

The design goals for the protocol are as follows:

- *Scalability:* the protocol must scale to networks of very large size. To this end, the protocol must ensure that the load on nodes and links is small and evenly distributed. In addition, for practical network topologies, the maximum processing load on each node and the maximum traffic load on each link should increase sublinearly with the network size.
- *Accuracy:*the accuracy requirement is subdivided into the following
  - (Soundness) An NTCA is raised only if the aggregate crosses T_g^+, and cleared only if the aggregate falls below T_g^-;
  - (Accuracy) An NTCA is raised if the upper threshold T_g^+ is exceeded for a certain minimal duration Δt_{alert}. For clearing the NTCA, the condition is symmetric;
  - (Timeliness) If an NTCA is raised (cleared), then it is raised (cleared) within some given minimal time t_{delay} after the relevant threshold crossing.
- *Robustness:* the protocol must adapt gracefully to changes in the underlying network topology, including node and link failures. This means that, for practically relevant scenarios involving node failures and/or topology changes, the protocol must produce output that is of practical use.

The Δt_{alert} condition is needed in order to adequately disregard transient behavior. A strict solution that does not allow for some such form of temporal imprecision cannot in fact be realized (cf. [12]). The soundness and timeliness conditions need to be interpreted in a similar way. We turn to this issue in section 7.

4 The GAP Protocol

The TCA-GAP protocol introduced in this paper is based on an earlier protocol, GAP - Generic Aggregation Protocol [3], for building and maintaining aggregation trees. GAP is a modified and extended version of the BFS (Breadth First Spanning) tree construction algorithm of Dolev, Israeli, and Moran [13]. The protocol of [13] executes in coarsely synchronized rounds, where each node exchanges with its neighbors its belief about the minimum distance to the root and then updates its belief accordingly. Each node also maintains a pointer to its parent, through which the BFS tree is represented.

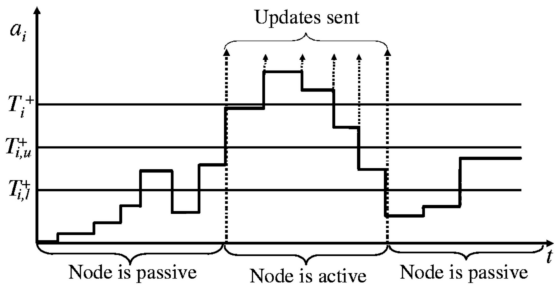
The above work by Dolev et al. [13] exhibits similarities to the 802.1d Spanning Tree Protocol (STP) [14, 15]. STP is a distributed protocol that constructs and maintains a spanning tree among bridges/switches, in order to interconnect LAN segments. Similar to [13], a node in STP chooses its parent, such that its distance (measured in aggregate link costs) to the root node is minimized. The initialization phase though is very different between the two protocols. While STP uses broadcast in LAN segments and a leader election algorithm to determine the root node, [13] assumes a given root node and an underlying neighbor discovery service. Also the failure discovery mechanism is very different in both protocols.

GAP extends [13] in a number of ways. First, GAP uses message passing instead of shared registers. Second, in GAP, each node maintains information about its children in the BFS tree, in order to compute the partial aggregate, i.e., the aggregate value of the weights from all nodes of the subtree where this node is the root. Third, GAP is event-driven. That is, messages are only exchanged as results of events, such as the detection of a new neighbor, the failure of a neighbor, an aggregate update, a change in local weight or a parent change. Fourth, since a purely event-driven protocol can cause a high load on the root node and on nodes close to the root, GAP uses a simple rate limitation scheme, which imposes an upper bound on message rates on each link.

In GAP, each node maintains a neighborhood table shown in Fig. 2(a), associating a status, a level, and an (aggregate) weight to each neighboring node. The status field (with values self, child, parent and peer) defines the structure of the aggregation tree. The value peer denotes a neighbor in the network graph that is not a neighbor in the aggregation tree. The level field indicates the distance, in number of hops, to the root. It is used to construct the BFS aggregation tree, whereby each node chooses its parent in such a way that its level is minimal. The weight field refers to the cached partial aggregate for a neighboring node and to the local weight for the local node.

Node	Status	Level	Weight
n_1	child	4	312
n_2	self	3	411
n_3	parent	2	7955
n_4	child	4	33
n_5	peer	4	567

(a) Sample neighborhood table for GAP



(b) Local hysteresis mechanism in TCA-GAP that decides whether a node is active or passive.

Fig. 2. The GAP neighborhood table and the local hysteresis mechanism in TCA-GAP

GAP relies on underlying failure and neighbor discovery services, which are assumed to be reliable.

5 TCA-GAP: A Distributed Solution to the NTCA Problem

TCA-GAP assumes a designated root node to report NTCAs. Upon starting the protocol, the root node will map the global thresholds into local thresholds for its children on the aggregation tree, and each child will then, recursively, assign local thresholds to its own children. During the operation of the protocol, each node that has an aggregate far below (or above) the local threshold will enter a passive state, where it refrains from forwarding changes of its aggregate to its parent. Once a node's aggregate gets close to its local threshold, it will become active and start reporting changes of the aggregate to its parent. A passive node will adapt to changes in network state and to failures, by dynamically recomputing the local thresholds of its children. In the following, we describe the main features of TCA-GAP in more detail.

Local hysteresis mechanism: a local hysteresis mechanism determines whether a node sends updates of its aggregate to its parent. A node that sends updates is called *active*. One that does not send updates is called *passive*. The local hysteresis mechanism is similar to the global hysteresis mechanism (see Fig. 1), but it serves a different purpose, namely, to correctly sample the aggregate when the global threshold is crossed. The local threshold assigned to node i is T_i^+. From T_i^+ the upper and lower local hysteresis thresholds $T_{i,u}^+$ and $T_{i,l}^+$ are computed, such that $T_{i,u}^+ = k_1 T_i^+$ and $T_{i,l}^+ = k_2 T_i^+$. Here, k_1 and k_2 are some *global control parameters*. Using these threshold values, a node will decide when to send its aggregate value as shown in Fig. 2(b). The transition between active and passive states occurs as follows. When the local aggregate of a passive node grows beyond the upper threshold $T_{i,u}^+$, the node becomes active. It will stop performing threshold recomputations (see below), start sending aggregate updates (just as in the GAP protocol), and reset the thresholds of its children to 0. When the aggregate of an active node decreases below $T_{i,l}^+$, then the node becomes passive and recomputes the thresholds of its children. The threshold of child j with aggregate a_j will be set to $T_i^+ * (a_j/a_i)$, where a_i is the aggregate of the local node.

Threshold rules: the (local) threshold rules guarantee that, if a global threshold is crossed, then there is at least one node, for which these rules are violated. They are:

(1) $T_i^+ \geq \Sigma_{j \in J} T_j^+$ where J is the set of children of node i;

(2) If J' is the set of active children, then $\Sigma_{j \in J'} T_j^+ \geq \Sigma_{j \in J'} a_j$ where a_j is the local aggregate reported by child j.

As long as these two rules remain valid on a node, it stays passive. Once one of them is violated, the node will recompute the thresholds of its children.

Threshold recomputation: threshold recomputation allows an active node to "receive threshold space" from one or more passive siblings. The purpose of this is to reinstate the threshold rules on the node.

It is generally difficult to recompute thresholds with a small overhead. For instance, a greedy approach to threshold recomputation will attempt to give an active child enough threshold space to make it passive. Such a solution, however, is prone to oscillation, since it can lead to two children alternately borrowing threshold space from each other. For this reason, we take a conservative approach that allows an active child of a passive node, under certain conditions, to remain active, without threshold recomputation having to occur.

Recomputation of local thresholds can happen for two reasons:

- Event #1: a node receives from its parent a new, lower threshold T', causing threshold rule (1) above to fail;
- Event #2: a child reports an increased aggregate, causing threshold rule (2) to fail.

These events can have several causes. For instance, the change of a local weight in some subtree can cause event #2 at the root of that subtree. When a device fails or is removed from the network, the topology of the aggregation tree may change, which in turn may cause events #1 or #2 to occur at different nodes in the network. The same can happen in the case where a device recovers from failure or is added to the network.

As a reaction to one of the above events, a node recomputes the local thresholds as follows:

i. For event #1, we reduce the threshold of one or more passive children by $\Sigma_{j\in J}T_j^+ - T'$, where J is the set of children of the current node. Observe that, if such a reduction is not possible, then T' will be less than the sum of the thresholds of the active children, and, therefore, has reverted to active state.
ii. For event #2, we reduce the threshold of one or more passive children by some value $\delta > \Sigma_{j\in J'}a_j - \Sigma_{j\in J'}T_j^+$, where J' is the set of active nodes, and increase the assigned threshold value of an active child by the same amount.

In case [ii.], there is some freedom in choosing δ and the child j whose threshold we increment. However, δ should not exceed $\frac{a_j}{k_2} - T_j^+$, since, as we noted, there is risk for oscillation. For our protocol, we choose j such that $a_j - T_j^+$ is maximized, and we choose $\delta = \frac{a_j}{k_1} - T_j^+$. Observe that, since the threshold rules are evaluated at the end of each protocol cycle, only an aggregate update from a single child can have caused event #2. Therefore, some j can be found, such that the resulting δ will satisfy [ii.]. To reduce the threshold of the passive children by δ, the threshold of each passive child is reduced, in turn, in the order of decreasing thresholds. This solution attempts to minimize the threshold updating overhead at the cost of a substantial risk that at least one child will become active in the next round.

Topology changes and failures: when a node is removed or fails, the protocol follows the GAP design, i.e., the failure detector informs all neighbors, and, as a result, the parent updates its neighborhood table by removing the failed node and recomputing its aggregate accordingly. When a new node is discovered, or a failed node recovers, the protocol again follows the GAP design by attaching it to a suitable parent. The parent receives an update message from the new node, creates an entry for the node in the neighborhood table, and updates the aggregate accordingly. In all of the above cases, the threshold rules are evaluated at the end of the protocol cycle, which may include threshold recomputation, etc., as described above.

Symmetric modes: once the aggregate exceeds T_g^+, then all nodes will have become active, and the overhead of TCA-GAP increases to that of GAP. To a large extent, this problem of a large overhead can be addressed by exploiting the inherent symmetry in the NTCA problem: detecting an upwards crossing of an upper threshold level is not different from detecting a downwards crossing of the lower one. In the first case, nodes will be passive on small aggregates and set to trigger when aggregates become large. In the second case, nodes will be passive on large aggregates and trigger when aggregates become small. Reflecting this, the protocol can work in one of two symmetric *modes*, positive or negative, depending on which threshold and which direction of threshold crossing it is set to trigger. The switch between modes is done at the root, as a result of global threshold crossings, and propagated down the aggregation tree, by adding the

mode to the update messages exchanged between neighbors. Locally, each node i is assigned either an upper threshold T_i^+ or a lower threshold T_i^-. In positive mode, the objective is to detect upwards crossings of T_i^+. In negative mode, the objective is symmetric, i.e., to detect downwards crossings of T_i^-. The local hysteresis thresholds in the latter case are computed as $T_{i,u}^- = T_i^-/k_1$ and $T_{i,l}^- = T_i^-/k_2$. Note that, for simplicity of presentation, the discussion in the previous subsections refers to the positive mode. The extension to negative mode is straightforward.

Initialization: the protocol initializes in the same way as GAP, which constructs the BFS spanning tree and populates the local neighborhood table [3]. As part of this initialization process, the local thresholds in all nodes are set to 0 in positive mode, causing weight changes to be reported up the aggregation tree to the root node. In the second phase of the initialization process, the root node sets its two global thresholds as instructed by the management station, which causes the recomputation of local thresholds to be propagated from the root down the aggregation tree. As a result, nodes start filtering weight and aggregate changes.

Code: the main data structure manipulated by TCA-GAP is the neighborhood table, which, in addition to the four columns of Fig. 2(a), contains a fifth column for thresholds. All numerical fields in the neighborhood table are arbitrarily initialized to 0. The update vector in TCA-GAP serves a similar purpose as in GAP, namely, informing a neighbor of a node about changes in the node's neighborhood table. This vector is a tuple of the form `(update,From,Weight,Level,Parent,ThresholdList, Sign)` where `ThresholdList` is a list of (node, threshold) pairs, and `Sign` is the mode. The protocol assumes an external, virtual root with level 0. Further, it assumes underlying services for failure detection and neighbor discovery. Local weight changes are handled by assuming that two instances of TCA-GAP run on every node, one, a leaf, for local weight changes, and one for aggregation. The main operation embodying the TCA-GAP semantics is the function `restoreTableInvariant`, which is responsible for maintaining the TCA-GAP invariants. These invariants ensure, for instance, that each node has a unique parent, and that the local threshold rules hold. In case the invariants are violated, actions are performed, such as selecting a new parent, switching between passive and active operation, or recomputing thresholds. The pseudo code for TCA-GAP can be found in [1].

6 Experimental Evaluation

We evaluated the functionality and performance of TCA-GAP through simulation, under varying topologies, thresholds, weight change and node failure models. The main hypotheses we wanted to test were:

- At low aggregate/threshold ratios the TCA-GAP scheme produces a management traffic overhead several orders of magnitude below that of a scheme for continuous monitoring such as GAP;
- The performance of TCA-GAP degrades gracefully as the aggregate/threshold ratio approaches 1.

The simulation results we have obtained are very encouraging. In the paper, we show results for two scenarios using an ISP network as the underlying topology. The first scenario involves several sinusoidal threshold crossings and shows how TCA-GAP successfully manages to reduce traffic when the aggregate is far from the thresholds. The second scenario shows the behavior of TCA-GAP at a low static aggregation level.

For the simulations we used a 221 node grid network topology and the topology of an ISP, Abovenet, from [16]. In this paper we mainly report on simulation results from Abovenet, a network with 654 nodes and 1332 links. The simulation studies were conducted with the SIMPSON network simulator [17]. The experiments were run with a uniform message size of 1Kbyte, a bandwidth of 100MB/sec, a processing delay of 1ms, and a communication delay of 4ms.

In the simulation runs, rate limitation was ignored to better compare the key properties of TCA-GAP versus the other two schemes. The main effect of rate limitation is to smooth peak traffic volumes, by imposing an upper bound on the traffic on each link. Secondly, the overall traffic is reduced since each node has the ability to process several messages before an output is produced.

Local weights are constrained to the interval $[0, \ldots, 100]$. Weight changes are simulated using a random walk model with random step sizes. Changes occur at randomly selected nodes, following a Poisson distribution with an average change rate per node of 1 weight change per second.

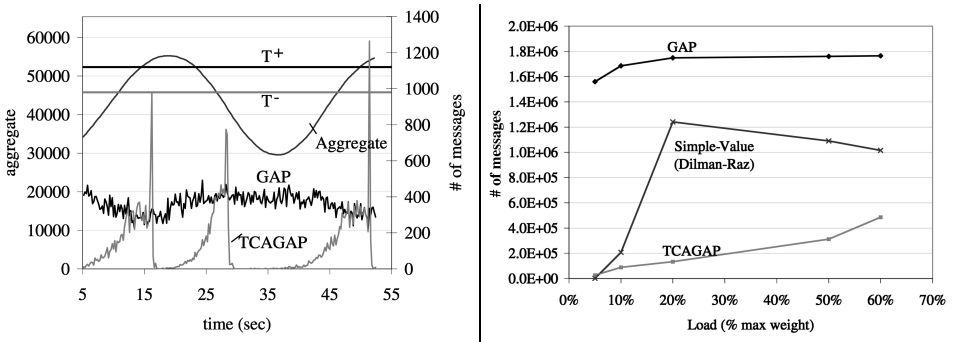
For the simulations shown in this paper, the global thresholds have been chosen at $T_g^+ = 80\%$ and T_g^- = 70% of the maximally achievable aggregate, i.e., the aggregate value where all nodes have the maximum possible weight of 100.

In the first scenario, nodes are initialized with weight values that are uniformly distributed in $[0, \ldots, 100]$ and the step size of the random walk model is biased with a sinusoidal input, such that $w_i(t + \Delta t) = w_i(t) + \Delta w + b * sin(t/\omega) + k$ where Δw is chosen to be uniformly distributed in an interval $[-x, \ldots, x]$, and the constants b and ω are chosen to obtain a suitable period and amplitude for the superimposed sinusoid on the aggregate. The constant k is a bias added to tune the aggregate to a desired long term average value. In Fig. 3(a), we show the aggregate, upper and lower global thresholds and, for both GAP and TCA-GAP, the total number of messages over a 200ms sampling period. The local thresholds are computed using $k_1 = 0.9$ and $k_2 = 0.85$. Three threshold crossings are shown in the figure. Each of them involves a gradually increasing number of active nodes, until a point is reached where all nodes are active. In the plot shown, this state is retained for a couple of seconds, until, after mode switching, the root reverts to passive state by crossing the relevant lower local threshold. (Note that, for any given node i in negative mode, the 'lower' threshold $T_{i,l}^-$ is actually higher than the 'higher' threshold $T_{i,u}^-$.) During the interval where all nodes are active, TCA-GAP behaves roughly as GAP. The large peaks are due to the root node propagating new threshold values, along with the sign, down the aggregation tree.

An interesting feature of TCA-GAP which is brought out in Fig. 3(a) is that traffic is concentrated around the global threshold crossings, and *not* where aggregates are maximal. This is attractive, since large aggregates are often indicative of congestion or overload situations in the network where it is desirable to keep management overhead to a minimum.

In the second scenario, the average aggregate weight has been chosen to be well below the global threshold, at about 5%. For modeling the weight changes, we used a biased version of the random walk model, such that $w_i(t + \Delta t) = w_i(t) + \Delta w + k$, where Δw is chosen to be uniformly distributed in an interval $[-x, \ldots, x]$.

The simulation results show that, on average, TCA-GAP generates 1.7% of the messages that are generated by GAP. At any given time, 95% of the nodes were passive and therefore did not produce messages. Graphs of the simulation results can be found in [1].



(a) Scenario 1: Messages generated by TCA-GAP and GAP; global aggregate over time; upper and lower values of global thresholds

(b) Total number of messages produced during a simulation run for TCA-GAP, SV and GAP in function of the average global aggregate.

Fig. 3. Simulation Results

For the second scenario, we performed similar simulation studies with higher variability of the weight changes. As expected, the traffic and processing overhead of TCA-GAP was larger than above, but still substantially smaller than that of GAP.

For both of the above scenarios, we performed the same simulations for a 221 node grid network. The simulation results were similar to those for the Abovenet topology, used in scenarios 1 and 2.

We have also compared our scheme to other schemes for detecting threshold crossing alerts [4] in the number of messages consumed in the whole network. To compare the performance of TCA-GAP with the simple-value scheme (SV) by Dilman-Raz [4], we performed a number of simulations using scenario 2 with various average aggregates. Fig. 3(b) shows the results of the simulations. It gives the number of messages generated by TCA-GAP, SV and GAP, as a function of the average aggregate during the simulation run of 900secs. As one can see, in GAP, the number of messages is hardly affected by the level of the aggregate. We explain this by the fact that, this protocol does not attempt any filtering and propagates all changes, no matter how small. The SV scheme by Dilman and Raz distributes static thresholds to all nodes, and determines whether a global TCA has occurred by polling all nodes whenever a node reports crossing of its locally assigned threshold level. From the measurements, we conclude that, for aggregate levels of less than 5%, SV produces a low number of messages compared to both GAP and TCA-GAP, since local thresholds are almost never crossed. However, as the aggregate level grow larger than 10%, the probability of local threshold crossings increases significantly, which is reflected by a large increase in traffic volume. The subsequent reduction in traffic for SV is due to the fact that, in SV, nodes sending traps do not subsequently need to be polled. For TCA-GAP the traffic volume increases at a much smaller rate.

7 Discussion and Future Work

In this section, we evaluate our simulation results against the design goals of TCA–GAP concerning accuracy, scalability and robustness set out in section 3, and point to areas of future work.

Accuracy: as we have pointed out, a protocol that does not allow some temporal imprecision cannot be engineered. In particular, in a practical system, threshold crossings

must have some minimum duration Δt_{alert} to guarantee detectability. Moreover, for a deterministic solution such as TCA-GAP, threshold detection can only be guaranteed when the network is stable: it is not hard to come up with failure patterns that conspire to continuously reconfigure the aggregation tree, such that threshold crossings never get detected at the root. For a stable network an upper bound on Δt_{alert}, the amount of time a threshold must remain crossed for TCA-GAP to guarantee detection, is $\mathcal{O}(n * (t_d + t_l) * h)$ where:

- n is the degree of the network graph = maximal number of neighbors for any node;
- t_d is the rate limitation timeout;
- t_l is the maximum communication delay between nodes;
- h is the height of the aggregation tree = the network diameter.

This bound is obtained, since, in the worst case, the threshold crossing has to be propagated to the root from the furthest distant leaf. Along each node, threshold violation has to be propagated through all neighbors in turn. Observe that the parameters n and h are determined by the choice of the management overlay topology, which, for this paper, is chosen to be identical to the underlying physical network topology. *Soundness*, in that NTCA's are raised only if the aggregate actually crosses T_g^+, is conjectured to hold in similar terms: if an NTCA is raised for at least the duration Δt_{alert} then it can be guaranteed that the threshold was also crossed some time during this interval. The *timeliness* requirement needs to be similarly adapted.

Scalability and Robustness: TCA-GAP uses a simple rate limitation scheme to impose an upper bound on the management traffic on each link. This by itself is not sufficient to guarantee scalability, however, unless a bound on the degree of nodes is also imposed. The effect of this is not trivial, and left for future work. For robustness, the protocol is certainly capable of adapting to topology changes in a graceful way. We confirmed this in simulations using scenarios with node failures and recoveries with results consistent with the other results we have reported.

Aggregation functions: above,we have used SUM as the aggregation function. Other simple aggregates like COUNT, MIN, and MAX can be supported with no modifications other than replacement of the aggregation function. For instance, to count the number of nodes with some local attribute exceeding c the local weight function $w_i(t)$ will return 1, if the attribute value exceeds c and 0 otherwise, and the aggregation function will be SUM. One way of handling AVERAGE in our framework could be to extend the underlying tree management protocol to maintain also node counts. This can be done by adding a further "tree size" field to the neighborhood table. Moreover, the cost of maintaining node counts only amounts to extending the topology update messages which are already exchanged by a node count field.

Future work: for the evaluation of our protocol in this paper, we have used a random walk model to capture the fluctuations of the device variables, i.e., the local weights. While random walk models have been used before to model the changes of device variables (e.g., changes in load on host interfaces [10]), we plan to further evaluate TCA-GAP using real traces. Further, we plan on analyzing the effect of using different overlay topologies on the performance of TCA-GAP and on making our protocol resilient with regard to root failures. Also, we plan to investigate proactive threshold recomputation schemes and more complex aggregation functions. Finally, an implementation of TCA-GAP on the Weaver platform [18] is under way in our laboratory at KTH.

Acknowledgments: This work has been supported by a grant from Cisco Systems and a personal grant from the Swedish Research Council.

References

1. F. Wuhib M. Dam R. Stadler and A. Clemm. Decentralized computation of threshold crossing alerts. Technical report, KTH Royal Institute of Technology, 2005.
2. S. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong. TAG: A Tiny AGggretation service for ad-hoc sensor networks. In *Proc. 5th Symposium on Operating Systems Design and Implementation*, pages 131–146, 2002.
3. M. Dam and R. Stadler. A generic protocol for network state aggregation. In *Proc. Radiovetenskap och Kommunikation (RVK)*, 2005.
4. M. Dilman and D. Raz. Efficient reactive monitoring. *IEEE Journal on Selected Areas in Communications (JSAC)*, 20(4), 2002.
5. David Breitgand, Danny Dolev, and Danny Raz. Accounting mechanism for membership size-dependent pricing of multicast traffic. In *Networked Group Communication*, pages 276–286, 2003.
6. R. van Renesse. The importance of aggregation. In *In (A. Schiper, A.A. Shvatsman, H. Weatherspoon, and B. Y. Zhao, eds.), Future Directions in Distributed Computing, Lecture Notes in Computer Science*, volume 2584, pages 87–92. Springer-Verlag, 2003.
7. I. Gupta, R. van Renesse, and K. Birman. Scalable fault-tolerant aggregation in large process groups. In *Proc. Conf. on Dependable Systems and Networks*, pages 433–442, 2001.
8. David Kempe, Alin Dobra, and Johannes Gehrke. Gossip-based computation of aggregate information. In *FOCS '03: Proceedings of the 44th Annual IEEE Symposium on Foundations of Computer Science*, page 482, Washington, DC, USA, 2003. IEEE Computer Society.
9. Mohamed A. Sharaf, Jonathan Beaver, Alexandros Labrinidis, and Panos K. Chrysanthis. Tina: a scheme for temporal coherency-aware in-network aggregation. In *MobiDe '03: Proceedings of the 3rd ACM international workshop on Data engineering for wireless and mobile access*, pages 69–76, New York, NY, USA, 2003. ACM Press.
10. Chris Olston, Jing Jiang, and Jennifer Widom. Adaptive filters for continuous queries over distributed data streams. In *SIGMOD '03: Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, pages 563–574, New York, NY, USA, 2003. ACM Press.
11. N. Roussopoulos A. Deligiannakis, Y. Kotidis. Hierarchical in-network data aggregation with quality guarantees. In *Proc. 9th International Conference on Extending Database Technology (EDBT)*, March 2004.
12. M. Bawa, H. Garcia-Molina, A. Gionis, and R. Motwani. Estimating aggregates on a peer-to-peer network. In *Manuscript*, 2003.
13. S. Dolev, A. Israeli, and S. Moran. Self-stabilization of dynamic systems assuming only read/write atomicity. *Distributed Computing*, 7:3–16, 1993.
14. IEEE. *ANSI/IEEE Std 802.1D, 1998 Edition*. IEEE, 1998.
15. R. Perlman. *Interconnections, Second Edition*. Addison Wesley Longman, 2000.
16. N. Spring, R. Mahajan, and D. Wetherall. Measuring ISP topologies with Rocketfuel. In *Proc. ACM/SIGCOMM*, 2002.
17. K. S. Lim and R. Stadler. SIMPSON — a SIMple Pattern Simulator fOr Networks. http://www.comet.columbia.edu/adm/software.htm, 2005.
18. K. S. Lim and R. Stadler. Weaver — realizing a scalable management paradigm on commodity routers. In *Proc. 8th IFIP/IEEE Int. Symp. on Integrated Network Management (IM 2003)*, 2003.