Computer Networks 52 (2008) 1745–1761

Contents lists available at ScienceDirect

Computer Networks

journal homepage: www.elsevier.com/locate/comnet

Decentralized detection of global threshold crossings using aggregation trees

Fetahi Wuhib, Mads Dam, Rolf Stadler *

ACCESS Linnaeus Center, Department of Electrical Engineering, KTH Royal Institute of Technology, 10044 Stockholm, Sweden

ARTICLE INFO

Article history:
Received 6 August 2007
Received in revised form 19 February 2008
Accepted 26 February 2008
Available online 14 March 2008

Responsible Editor: Prof. J. Neuman de Souza

Keywords:
Decentralized network management
Threshold crossing alerts
Real-time monitoring
Tree-based aggregation protocols

ABSTRACT

The timely detection that a monitored variable has crossed a given threshold is a fundamental requirement for many network management applications. A challenge is the detection of threshold crossing of network-wide variables, which are computed from device counters across the network, using aggregation functions such as SUM, MAX and AVERAGE. This paper contains a detailed description and a comprehensive evaluation of TCA–GAP, a protocol for detecting threshold crossings of network-wide aggregates in a distributed way. Elements of its design include tree-based incremental aggregation for estimating the value of aggregates, a local hysteresis mechanism to reduce overhead and dynamic recomputation of local thresholds to ensure correctness. The protocol is evaluated through extensive simulation using real traces in scenarios with network sizes up to 5232 nodes. From the measurements, we conclude that the protocol is efficient in the sense that the overhead is negligible when the aggregate is far from the threshold. It is scalable as the protocol overhead is independent of the system size for the network sizes and scenario configurations considered. We demonstrate that the local hysteresis parameter can be used to control the tradeoff between protocol overhead and detection delay. We further report on results on how node failures impact overhead and detection quality of the protocol.

1. Introduction

Threshold crossing alerts (TCAs) indicate to a management system that a monitored management variable or MIB object 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 T^{g+} is typically accompanied by a second threshold T^{g-} 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 (see Fig. 1).

The TCA concept enables a management activity to be event based, instead of relying on centralized polling. This results in management applications that scale significantly better, in addition to being more responsive, as they do not incur the delay that is associated with polling cycles.

Today, TCAs are generally configured per device, e.g. for monitoring levels of utilization or packet drop rates on a particular link. Similarly, service level agreements (SLAs) are often articulated in terms of parameters that can be monitored on a per device level, reflecting today's technical limitations. However, there are many cases where cross-device TCAs are essential, whereby local variables are aggregated across the network or across a network domain. Examples include thresholds on network-wide performance parameters, such as the average link utilization, the current level of p2p traffic, or the number of VoIP flows in a network domain.

The hard part in determining when to trigger a network-wide TCA is to ensure scalability and fault tolerance

* Corresponding author. Tel.: +46 8 790 4250.
E-mail addresses: fetahi@kth.se (F. Wuhib), mfd@kth.se (M. Dam), stadler@kth.se (R. Stadler).

1389-1286/$ - see front matter © 2008 Elsevier B.V. All rights reserved.
doi:10.1016/j.comnet.2008.02.015

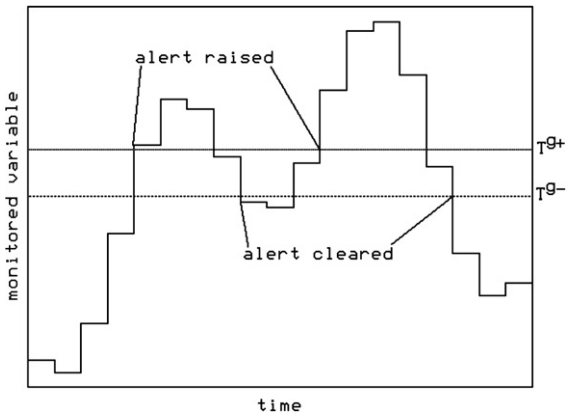

Fig. 1. Threshold crossing alerts: an alert is raised when the monitored variable crosses a given threshold T^{g+} from below. The alert is cleared when the variable crosses a lower threshold T^{g-} from above.

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 local 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.

This work focuses on *detecting threshold crossings of network-wide aggregates in a distributed way*, without the need for a special coordinating node. To this end, we assume that each network device participates in the monitoring activity 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).

A straightforward approach to detecting threshold crossing of network-wide aggregates would be to use a protocol for distributed state aggregation, e.g. [4,5]. Such a protocol provides 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. Threshold crossings can then be detected at the root node by continuously evaluating the aggregate against the threshold. As we show in this paper, however, such an approach fails to benefit from possible reduction in overhead when the aggregate is far from the threshold.

In this paper, we present TCA–GAP, a tree-based protocol for detecting TCAs on network-wide aggregates of local variables. Aggregation functions the protocol supports include SUM, AVERAGE, COUNT, MAX and MIN. The detection process is performed in a distributed way, and it dynamically adapts to the network state. The protocol has negligible overhead if the aggregate is sufficiently far from the threshold. It is robust to node and link failures. It is scalable in network size with respect to protocol overhead and threshold detection time. Lastly, the tradeoff between protocol overhead and detection time can be controlled. The basic concepts behind the protocol are tree-based, incremental aggregation for estimating the global and partial aggregates. Local thresholds and a local hysteresis mechanism switch nodes to a passive state whenever their contribution to threshold detection is not needed. A local mechanism for dynamic recomputation of local thresholds, triggered by violation of local invariants, ensures the detection of threshold crossings.

The paper contains a detailed description of the protocol, including a formalization of invariants and correctness statements, a discussion of different policies for threshold recomputation and the pseudocode for key parts of the protocol. In addition, it includes a comprehensive study of the protocol's performance in simulation scenarios using real traces and system sizes up to 5232 nodes. Specifically, the protocol's efficiency, scalability and robustness properties are evaluated. It is demonstrated that the tradeoff between protocol overhead and detection delay can be controlled. As part of reviewing related work, the paper compares the design and performance of the protocol with that of HRMA [18], a protocol with similar purpose and design characteristics.

The paper reports on results from our work on threshold detection which has progressed over several years and produced several intermediate results [6–8]. The idea for this protocol has first been presented in [6]. In this paper, we gave evidence that the concept of local thresholds and threshold recomputation can be realized using tree-based aggregation. We showed, through simulation using synthetic traces that such a protocol can be efficient, compared to the naïve approach discussed above. In [7], we reported on a testbed implementation and measurements, which confirmed the simulation results. In the magazine article [8], the focus is on motivating threshold crossing alerts for network-wide aggregates and high-level description of the protocol.

The contribution and originality of this paper lies, first, in a complete and detailed description (as well as improved and extended version) of the protocol outlined in [6], including some formalization and the pseudocode. Second, we provide a comprehensive evaluation of the protocol using real traces, and specifically report results on scalability and robustness for the first time. Third, we show how the tradeoff between overhead and detection delay can be controlled through a protocol parameter. Fourth, we include a comprehensive review of related work and a comparison with HRMA.

The paper is organized as follows. Section 2 presents the objective and the design goals of the protocol we present in this paper. Section 3 presents our protocol, TCA–GAP. Section 4 presents the results of the experimental evaluation. Section 5 presents a review of related work. Finally, section 6 concludes the work.

2. Objective and protocol design goals

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 local state variable $w_i(t)$ (sometimes called weight) that represents the quan-

tity whose aggregate is being subjected to threshold monitoring. For the remainder of this paper, we assume that local variables are *non-negative real-valued* quantities, aggregated using SUM. The objective is to raise an alert on a distinguished root node, when the aggregate local variable $\sum_i w_i(t)$ exceeds a given global threshold T^{g+}, and to clear the alert when the aggregate has decreased below a lower threshold T^{g-} (see Fig. 1).

We engineer a protocol that realizes the objective described above with the following design goals.

- *Efficiency*: The communication and processing overhead of the protocol should be small, specifically during periods where the aggregate is far from the threshold (above or below).
- *Quality of detection*: The protocol should achieve small delays for detecting threshold crossings, and false positives and false negatives should be extremely rare.
- *Scalability*: The protocol should allow for efficient operation with high quality of detection in large networks with at least thousands of nodes.
- *Robustness*: The protocol should allow for a continuous operation after node or link failures, as well as after addition and removal of nodes.
- *Controllability*: The protocol should allow for controlling the tradeoff between quality of detection and protocol overhead through management parameters that can be adjusted at runtime.

3. The protocol: TCA–GAP

3.1. Protocol overview

The threshold detection protocol we present in this paper can be seen as a non-trivial, significant extension of a tree-based aggregation protocol called GAP [4]. GAP creates and maintains a breadth first spanning tree, along which aggregation is performed. The protocol is robust to failures and to topology changes, and it is scalable, thanks to the use of a rate limitation scheme that prevents nodes near to the root from becoming overloaded.

The proposed protocol, which we call TCA–GAP, extends GAP in a number of ways. One extension is that the root node in TCA–GAP raises and clears alerts. TCA–GAP also allows a node in an *active* state, where it executes the GAP protocol, to enter a *passive* state, where it ceases to propagate updates up the aggregation tree. The transition between active and passive states is controlled by a local hysteresis mechanism. To achieve efficiency (i.e. low overhead), the protocol attempts to maximize the number of passive nodes through dynamic reallocation of local thresholds. This reallocation is governed by local invariants that ensure that threshold crossings will not get unreported. The above mechanism, which reduces overhead while ensuring the detection of threshold crossings, comes with some penalties. First, the switching of a node (or more precisely a subtree) from passive to active state introduces delays, which reduce the quality of threshold detection. Second, threshold reallocation introduces a certain amount of additional overhead. TCA–GAP allows controlling the tradeoff between protocol overhead and quality of detection through a parameter of the hysteresis mechanism.

Finally, we observe that there is a symmetry in the hysteresis model between upwards crossings of an upper threshold and downwards crossings of a lower threshold (see Fig. 1). To exploit this symmetry, we introduce the concept of dual modes: a "positive" mode in which the protocol detects the crossing of the upper threshold and a symmetric "negative" mode. Whenever the current global threshold is crossed, the root node switches mode and the new mode is propagated down the aggregation tree, together with new local thresholds. Generally, if the root node switches mode, it becomes passive, and consequently all other nodes become passive as well.

As a consequence of the design, the protocol overhead is usually concentrated to periods just before and just after threshold crossings.

For simplicity of presentation, the discussion of the protocol refers to detecting crossing of the upper threshold, i.e. is restricted to the positive mode. The extension to negative mode is straightforward.

3.2. The GAP protocol

The generic aggregation protocol (GAP) [4] allows for continuous monitoring of network-wide aggregates through the use of an aggregation tree. GAP is an adapted and extended version of the breadth first spanning (BFS) tree construction algorithm of Dolev et al. [9]. The protocol in [9] 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.

The above protocol by Dolev et al. exhibits similarities to the 802.1d spanning tree protocol (STP) [10]. STP is a distributed protocol that constructs and maintains a spanning tree among bridges/switches, in order to interconnect Ethernet segments. Similar to [9], 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, [9] assumes a given root node and an underlying neighbor discovery service. Also the failure discovery mechanism is very different in both protocols.

GAP extends [9] in a number of ways. First, GAP relies on message passing instead of shared registers. Second, in GAP, each node maintains a pointer to its parent, through which the BFS tree is represented. Third, 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 local variable from all nodes of the subtree where this node is the root. Fourth, 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 variable or a change in parent. Fifth, 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

Table 1
Neighbor table in GAP for node *c* in Fig. 2

Node ID	Role	Level	Partial aggregate
a	Parent	1	140
b	Peer	2	70
c	Self	2	10
f	Peer	3	10
g	Child	3	40
h	Child	3	10

rates on each link. Finally, GAP is more efficient communication-wise than [9], since it implements selective propagation of updates.

In GAP, each node maintains a neighborhood table shown in Table 1, associating a role, a level, and a partial aggregate to itself and the neighboring nodes on the network graph. The *role* field (with values self, child, parent and peer) defines the structure of the aggregation tree. The value peer denotes a neighbor on the network graph that is not a neighbor on the aggregation tree. The field *level* 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 *partial aggregate* field refers to the cached partial aggregate for child nodes and to the local variable for the local node.

Aggregation over the spanning tree is performed incrementally through local computations, where each node computes the partial aggregate of its subtree from the partial aggregates of its children and its own local variable. i.e. for a node i, the partial aggregate a_i is computed as $a_i = w_i + \sum_j a_j$ where j is a child of node i and w_i is the local variable of i. The value of the partial aggregate at the root node corresponds to the global aggregate of the aggregation tree. Fig. 2 shows a spanning tree created by GAP for this purpose.

A node communicates changes to its neighborhood table by sending an *update vector* to its neighbors. An update vector of a node contains information about its partial aggregate, its distance to the root node and the ID of its parent. Through this mechanism, the topology of the aggregation tree is maintained and changes to its local variable are propagated from the node to the root along a path on the aggregation tree.

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

3.3. Local threshold and hysteresis mechanism

A key idea in TCA–GAP is to introduce and maintain local thresholds that apply to each node in the aggregation tree. These local thresholds allow nodes to switch between an *active* state, where the node executes the GAP protocol and sends updates of its partial aggregate to its parent, and a *passive* state, where the node ceases to propagate updates up the aggregation tree. The transition between active and passive state is controlled by a *local threshold* and a *local hysteresis* mechanism.

(We restrict the discussion in Sections 3.3 and 3.4 to the case where the crossing of the upper global threshold is detected. For reason of readability, we do not write T^{g+}, but simply T^g. An extension of the discussion to detecting the lower global threshold is straightforward, as will be shown in Section 3.5.)

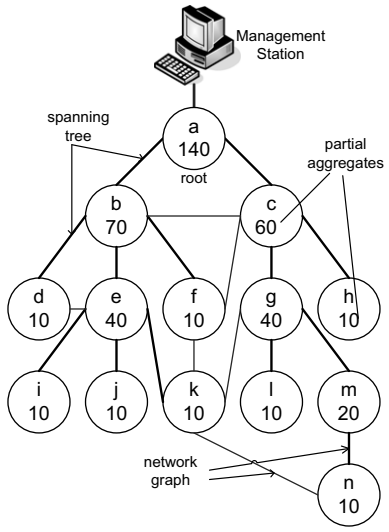

Fig. 2. GAP maintains a spanning tree (bold lines) on the network graph (bold and light lines) that enables incremental aggregation of local variables on a continuous basis. Each node shows its ID and the partial aggregate, for the aggregation function SUM. Local values are 10 for all nodes.

Each node has a local threshold, which is set by its parent. For the root node, the local threshold is the same as the global threshold T^g.

Two global parameters $0 \leqslant k_2 \leqslant k_1 \leqslant 1$ configure the local hysteresis mechanism and control how the node switches between active and passive state. We first explain the transition from active to passive. When a node is active, then all nodes in the subtree rooted at that node are also active (how this is achieved is explained below). The transition from active to passive state takes place whenever the partial aggregate $a_i(t)$ of a node i is less than $k_2 T_i$ (see Fig. 3). When the node turns passive, it assigns a local threshold T_j to each child j, proportionally to the partial aggregate $a_j(t)$ of the child j: $T_j = T_i \frac{a_j}{w_i + \sum_j a_j}$.

Second, a node i in passive state becomes active when it can deduce that its partial aggregate a_i must exceed $k_1 T_i^+$. It makes this deduction based on the fact that (a) the partial aggregate of each active child is known to the node and (b) the node knows that, for each passive child j, the partial aggregate a_j is below $k_1 T_j$.

When a node i switches to active, it sets the threshold of its children to 0. As a consequence, all nodes in the subtree rooted at node i have their local thresholds recursively set to 0 and, as a result, become active.

At the start of the protocol, the local thresholds and the hysteresis mechanisms are initialized as follows. All nodes start in active state with local threshold 0 and in positive mode (see section 3.5), which means that they execute the GAP protocol, i.e. the aggregation tree is constructed, the partial aggregates are computed on all nodes, and the estimate of the global aggregate is available at the root node. At the end of the initialization phase, the root node switches to passive state, sets its local threshold to the global (upper) threshold T^g and assigns local thresholds to its children proportional to their partial aggregates (see above).

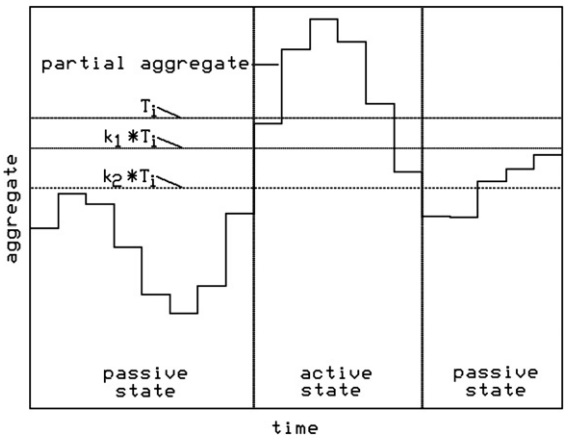

Fig. 3. The local hysteresis mechanism controls the switch between active and passive state of a node.

The parameters k_1 and k_2 of the hysteresis mechanism control the tradeoff between the protocol overhead and the quality of TCA detection. For small values of k_1 and k_2, the protocol tends to make nodes active when the aggregate is relative far from the threshold, hence increasing the overhead. On the other hand, the larger number of active nodes, the shorter the detection time tends to become. Values for k_1 and k_2 close to 1 cause the protocol to keep nodes passive unless the aggregate is close to the threshold. Though this decreases the protocol overhead, it generally increases the detection time, since less aggregate updates are reaching the root node.

We close with two comments on the choice of k_1 and k_2. First, k_2 must be chosen strictly smaller than k_1 to achieve the hysteresis behavior. Second, if $k_2 = k_1 = 0$, then the protocol exhibits the same behavior as GAP, since all nodes are kept active.

3.4. Local threshold rules and local threshold recomputation

TCA–GAP attempts to reduce the protocol overhead by keeping as many nodes passive as possible while ensuring threshold detection. To enable threshold detection at the root node, we introduce *local invariants* that must hold on passive nodes together with *threshold recomputation policies*, which are triggered upon violation of these invariants.

Consider a node i in passive state. There are two conditions under which node i may become active. First, it may be that i's parent reduces the local threshold T_i assigned to i (for the purpose of threshold recomputation; see below). The result may be that T_i becomes strictly smaller than $w_i + \sum_{j\in J} T_j$ with T_j being the threshold assigned to child j and J being the set of children of node i. This can cause the local threshold T_i to be crossed without this being detected at node i, resulting in the possibility of a false negative at the root node. To prevent such a scenario from happening, we introduce the following local threshold rule for node i:

$$\text{(R1)} \quad T_i \geqslant w_i + \sum_{j\in J} T_j,$$

where J is the set of children of node i.

The second condition under which node i might need to switch to active state concerns the situation where one or more of its children are active. Recall that the local hysteresis mechanism ensures that the actual aggregate of a subtree rooted in a passive child j does not exceed T_j (at least in the approximate sense as computed by the underlying aggregation protocol, GAP). Thus, a sufficient condition for the actual aggregate of i's subtree to not exceed T_i is that the sum of aggregates reported by active children does not exceed the sum of the corresponding local thresholds. This motivates the second local threshold rule:

$$\text{(R2)} \quad \sum_{j\in J'} T_j \geqslant \sum_{j\in J'} a_j(t),$$

where J' is the set of active children of node i.

Rules R1 and R2 together imply the following proposition, which ensures that local threshold crossings will be detected. ($\leqslant$ denotes the ancestral relation between nodes.)

Proposition 1. *Suppose that rules R1 and R2 hold for a passive node i.*

If $\sum_{j'\leqslant j} w_{j'}(t) \leqslant a_j(t)$ *whenever j is an active child of i, and* $\sum_{j'\leqslant j} w_{j'}(t) \leqslant T_{j'}$ *whenever j is a passive child of i, then* $\sum_{j\leqslant i} w_j(t) \leqslant T_i$.

Proof. Follows immediately from R1 and R2. □

Corollary. *Suppose that rules R1 and R2 hold for all passive nodes of the aggregation tree and that the root node is passive.*

If $\sum_{j'\leqslant j} w_{j'}(t) \leqslant a_j(t)$ *for all active nodes j, then* $\sum_i w_i(t) \leqslant T_g$.

Proof. Induction using Proposition 1.

Note that, if we assume no delays in computing partial aggregates on subtrees with active nodes, the above corollary implies that if R1 and R2 hold on all passive nodes, all threshold crossings will be detected at the root node.

If one of the rules R1 and R2 fails on node i, then the node attempts to reinstate the rule by reducing the threshold of one or more passive children. We call this procedure *threshold recomputation*. Specifically, if (R1) fails, then the protocol reduces the threshold of one or more passive children by $\delta = w_i + \sum_{j\in J} T_j - T_i$, where J is the set of children of i. Evidently, this may cause one or more passive children to become active.

If (R2) fails, then the protocol reduces the threshold of one or more passive children by $\delta > \sum_{j\in J'} (a_j - T_j)$ where J' is the set of active children, and, at the same time, increases the assigned threshold of one or more active children by the same amount, which will reinstate (R2). Such a reduction is always possible since the node is passive.

There are many possible policies for threshold recomputation. For instance, there are several ways to choose the set of active children whose threshold is increased. Note though that the amount of threshold increment for child j must not exceed $\frac{a_j}{k_2} - T_j$. If it does, there exists a scenario in which two children alternately borrow threshold space from each other and the system oscillates. In TCA–GAP the protocol identifies the smallest set of active chil-

dren j^* with the largest values of $\frac{a_j}{k_1} - T_j$ from all $j \in j'$ so that $\sum_{j \in J*} \left(\frac{a_j}{k_1} - T_j\right) > \sum_{j \in J'} (a_j - T_j)$. Then δ is chosen such that $\delta = \sum_{j \in J*} \left(\frac{a_j}{k_1} - T_j^+\right)$ and the threshold of a child j is increased by $\frac{a_j}{k_2} - T_j$ for all $j \in j^*$.

There are also options on how to choose the set of passive children whose threshold is reduced. We give three examples of such possible policies.

Policy I: The child j with the largest threshold T_j is selected. If $\delta \leqslant T_j$, then j is the only child whose threshold is reduced. Otherwise, T_j is reduced to 0, and this procedure is applied to the child with the second largest threshold and $\delta := \delta - T_j$. This policy attempts to minimize the overhead for threshold updating at the cost of increasing the risk of nodes becoming active. The simulation results in this paper are based on *Policy I*.

Policy II: The thresholds of all passive children are reduced at the same time, by an amount that is proportional to the threshold of each child. This policy attempts to minimize the risk of children becoming active, while allowing for a larger overhead for threshold updating.

Policy III: This policy is similar to the policy above in that all passive children are chosen for threshold reduction. However, their thresholds are set to 0, in effect making the entire subtree of the local node active. This policy increases the quality of threshold detection at the expense of a larger overhead.

3.5. Symmetric modes

After detecting an upward crossing of the upper threshold T^{g+} the task of threshold detection becomes that of detecting a downward crossing of the lower threshold T^{g-} (see Fig. 1). The protocol design captures this change by having TCA–GAP execute in one of two symmetric modes, *positive* or *negative*, depending on which threshold and which direction of threshold crossing it is set to detect. In the first case, nodes will be passive when aggregates are small, and the alarm is raised when the global aggregate becomes too large. In the second case, nodes will be passive when aggregates are large, and the alarm is cleared when the global aggregate becomes too small (see Fig. 1).

Whenever the protocol detects a global threshold crossing, the root node switches mode, and the new mode is propagated down the aggregation tree. If the thresholds and the control parameters are chosen such that $k_1 T^{g+} > T^{g-}$, then the root node becomes passive after it switches between modes (and possibly all other nodes become passive as well).

Recall that, in the positive mode, a node i becomes active when its partial aggregate exceeds $k_1 T_i$, and it switches to passive when its partial aggregate falls below $k_2 T_i$. In the negative mode, node i becomes active when its partial aggregate falls below T_i/k_1, and it switches to passive when its partial aggregate exceeds T_i/k_2.

3.6. Pseudocode

The main data structure in TCA–GAP is the neighborhood table, which extends the neighborhood table of the GAP protocol (see Table 1) with a fifth column for thresholds and a sixth one for state. The main message type is update vector, which is used to inform the neighbors of a node about changes in the neighborhood table. An update vector is of the form (`update`, `From`, `Weight`, `Level`, `Parent`, `State`, `ThresholdList`, `Mode`), where `From` identifies the sender, Weight is its partial aggregate, `Level` is its belief of its distance from the root node, `Parent` is its parent in the spanning tree, `State` is the current state of the node ('active' or 'passive'), `ThresholdList` is a list of (node, threshold)-pairs, and `Mode` is a binary variable with values positive or negative.

The protocol assumes underlying services (with associated message types) for failure detection (`fail`), neighbor discovery (`new`), change of local variable (`updateLocal`) and a timer (`timeOut`).

```
   proc tca-gap(k1, k2, Tg+, Tg-) = {
    /* initialization */
1   table=[(self(),self,0,0,0)];
2   timeOut=true; mode=pos; state=active;
3   localThreshold=0; initializeServices();
4   vector=updateVector(Table);
5   newVector=vector;
    /* protocol loop */
6   while (true) {
7     receive {
8       (new,From) =>
9         addEntry(From,table);
10    | (fail,From) =>
11        removeEntry(From,table);
12        if (From==parent())
13          localThreshold=0;
14    | (update,From,Weight,Level,Parent,
          State,thresholdList,Mode) =>
15        updateTable(table,From,Weight,Level,
            Parent,State,thresholdList,Mode);
16    | (updateLocal, Weight) =>
17        setLocalWeight(table,Weight);
18    | (timeout) =>
19         timeOut=true;
20    end; } /* end receive */
21    if (TCAGAPinitTimeout()) {
22      localThreshold=Tg+;
23     } /* end initialization */
24    if (thresholdCrossed()) {
25      raiseOrClearTCA(mode);
26      if (mode==pos) {
27        localThreshold=Tg-;
28        mode=neg;}
29      else {
30       localThreshold=Tg+;
31       mode=pos;}
32    } /* end thresholdCrossed */
33    restoreInvariants() ;
34    newVector=updateVector(table) ;
35    if newVector!=vector and timeOut {
36      sendToNeighbors(newVector,vector);
37      timeOut=false;
38      vector=newVector;
39    } /* end if vector changed */
40  } /* end while */
41 } /* end tca-gap */
```

Fig. 4. The main procedure of TCA–GAP.

The function `restoreInvariants()` is responsible for maintaining the protocol invariants. These invariants relate to the structure of the spanning tree, the local threshold rules and hysteresis mechanism. If an invariant is violated, the appropriate action is performed to reinstate it, e.g. selecting a new parent, switching from passive to active state, or recomputing the thresholds of children. The pseudocode for the main procedure `tca-gap()` is given in Fig. 4 and `restoreInvariants()` in Fig. 5. A summary of other key functions is given below.

- `updateVector()`: Generates the update vector.
- `TCAGAPinitTimeout()`: Returns true at the root node when TCA–GAP is initialized.
- `aggr()`: Returns the partial aggregate of the node as computed over the reported aggregate of active children.
- `thresholdCrossed()`: Returns true at the root node when the threshold is crossed.
- `raiseOrClearTCA()`: Sends alarms raise/clear signals to the management station.
- `sendToNeighbors()`: Sends the update vector to selected neighbors. If partial aggregate changes then parent node is included in the recipient list. If threshold of a node changes, that node is included. If parent or level changes, then all neighbors are included.
- `usedThresholds()`: Returns the sum of the threshold assigned to children.
- `reduceThresholds()`: Reduces the thresholds of selected passive children according to the policy used (Section 3.4).

```
   proc restoreInvariants() = {
   /* parent must have minimum level among neigh-
   bors */
1  ... /* consult pseudocode of GAP[4] */
   /* a node must have correct state, passive or
   active */
2   if localThreshold == 0
       or
      (mode==pos and aggr()>k1*localThreshold)
       or
      (mode==neg and aggr()<localThreshold/k1)
3   {
4     setThresholdsChildrenToZero();
5     state=active;
6     exit proc;
    }
7   if (state==active) and
      ((mode==pos and aggr()<k2*localThreshold)
       or
      (mode==neg and aggr()>localThreshold/k2))
8   {
9       setThresholdsChildren();
10      state=passive;
11  }
   /* Threshold rules R1 and R2 must hold */
12  if state==passive {
13    if (R1 is violated)
14    needed=|usedThreshold()- localThreshold|;
15    if (R2 is violated) {
16      find j* ;
17      needed=thresholdNeeded(j*);
18      incrementThreshold(j*);
19    }
20    reduceThresholds(needed); }
21  } /* end if */
22 }   /* end restoreInvariants */
```

Fig. 5. The `restoreInvariants()` function.

As can be seen from Fig. 4, TCA–GAP starts with initializing the neighborhood table `table` (line 1). For the root node, an additional entry of a virtual root with level-1 is added. All nodes initialize in positive mode and active state (line 2). The various services are initialized (line 3). Then, the protocol executes the GAP protocol until the initialization phase completes by a signal from `TCAGAPinitTimeout()` on the root node (line 21), at which point the root sets its local threshold to the global threshold and the operational phase of the protocol starts.

The protocol executes a loop (line 6–39) in which a node processes messages that it receives from local services and from neighbors (lines 7–20). The protocol processes the messages as follows.

- `(new, From)`: this message is sent to the TCA–GAP process by the neighbor discovery service when a new node is detected. The node is added to the neighborhood table (line 8).
- `(fail, From)`: this message is sent when the failure of a neighbor is detected. The entry of the failed node in the neighbor table is removed. If the failed node has been the parent, the local threshold is also set to 0. (lines 10–13).
- `(update, From, ...)`: this message contains an update vector from a neighbor with ID From. This causes the row corresponding to From in the neighborhood table to be updated. If From is the parent node, then localThreshold and mode are also updated.
- `(updateLocal, Weight)`: this message indicates a change in the local variable. The protocol updates its entry in the neighbor table accordingly.
- `(timeout)`: this message is sent by the timer service. It is used to control maximum message rate on overlay links.

After processing a message, the protocol performs the following operations. First, if the node is root, it checks whether the threshold is crossed (line 24). If it is, it sends an alert to the management station and switches mode (lines 24–32). Then, all nodes call `restoreInvariants()` to reinstate the protocol invariants, which may have been violated as a result of processing a message (line 33). Finally, the node computes a new update vector, which is sent to neighbors, if it differs from the last vector sent and the maximum message rate allows that (line 35–39).

The `restoreInvariants()` function in Fig. 5 tests and possibly reinstates the protocol invariants. First, it ensures that the current parent has the minimum level among all neighbors, in order to maintain a BFS spanning tree (line 1). Next, it checks whether a node needs to switch state from active to passive or vice versa. If the local threshold

is 0 or if the partial aggregate is larger than the upper hysteresis threshold (or smaller than the lower threshold in negative mode), then the node, if not already active, switches to active state and resets the threshold of its children to 0 (lines 2–5). If the node is active and the aggregate is below the lower hysteresis threshold (or above the upper hysteresis threshold in negative mode), then it switches to passive state and sets the threshold of its children (see Section 3.3). Finally, a passive node verifies the local rules and performs threshold recomputation if needed (lines 14–25).

4. Experimental evaluation

We have evaluated TCA–GAP on a testbed and through simulation. The evaluation on testbed has been performed on Weaver [11], a distributed management platform, and results have been reported in [7,8]. Simulation studies have been performed using SIMPSON [12], a discrete event simulator that allows us to simulate message exchanges over large network topologies and message processing on the network nodes. (The key reason for choosing SIMPSON in this study over one of the popular network simulators like NS2 has been its suitability for simulating large networks.) The simulation model we use is quite detailed in the sense that it captures the communication delays of messages on overlay links, processing delays and delays within processor queues, as well as topology changes including node and link failures. However it abstracts from lower-level networking issues, such as physical layer characteristics, packet loss and packet encapsulation.

In this paper, we present simulation results from various scenarios where we evaluate the protocol efficiency, the latency in threshold detection, scalability with respect to the number of nodes and the robustness of the protocol under various failure rates. In contrast to the preliminary simulation studies reported earlier, all simulation studies for this paper have been performed using real traces to simulate the local variables.

4.1. Simulation setup and evaluation scenarios

4.1.1. Evaluation metrics

We evaluate TCA–GAP using the following metrics. First, we measure the protocol overhead as the average number of messages processed/sec/node. Second, we evaluate the quality of TCA detection by measuring the correctness of the detection and the detection delay. The correctness of the detection is determined by (a) the ratio of false negatives (i.e. cases where the protocol fails to raise an alert although a threshold crossing has occurred) to the total number of threshold crossings and (b) the ratio of false positives (i.e. cases where the protocol raises an alert even though no threshold crossing has occurred) to the total number of alerts raised by the protocol. We measure the detection delay as the difference between the time TCA–GAP reports a crossing and the time the actual crossing occurs. In the graphs illustrating the measurement results, 95% confidence intervals are given wherever appropriate.

4.1.2. Local variables

In all scenarios, a local variable represents the number of HTTP flows that enter the network at a specific router, and the aggregate of those variables represents the total number of such flows in the network. We simulate the behavior of the local variables using packet traces captured at the University of Twente [13]. The first trace which we call the *UT trace* has been created as follows. We sampled every second the number of HTTP flows from those original traces, which produced traces that give the evolution of the number of HTTP flows over time. Then, we divided the new traces into segments of 150 s each. From those segments, we constructed traces of 1500 s for each node in the simulation, by randomly selecting and concatenating ten of those segments. Across all traces, the average value of the local variables is about 45 flows, and the standard deviation of the change between two consecutive samples is about 3.4 flows.

The second trace, which we call *Periodic UT trace*, is obtained by adding a sinusoidal bias to the UT trace $w_i(t)$ on a node i as follows:

$w_i(t)^* = int(w_i(t) + 23^*(1 + \sin(2\pi t/30 - \pi/2)))$ where int() returns the integer component of its argument. In our simulations, we use the UT trace to study the behavior of TCA–GAP in scenarios where no threshold crossing occurs, while we use the Periodic UT trace in scenarios where multiple threshold crossings occur.

4.1.3. Overlay topology

The topologies used for the network graphs in our simulations are generated by GoCast [14], a gossip protocol that builds topologies with bidirectional edges and small diameters. The protocol allows setting the (target) connectivity of the graph. For the measurements reported in this paper, we do not simulate the dynamics of GoCast. This means that the topology does not change during a simulation run. Unless stated otherwise, the topology used in the simulations has 654 nodes (this is the size of Abovenet, an ISP [19]). All topologies are generated with a target connectivity of 5, which, for the 654 node topology we use, produces an average inter-node distance of 4.3 hops and a diameter of 7 hops in the graph.

While in [6] we performed the evaluation of TCA–GAP using an overlay that mirrors the physical topology, in this study we apply an overlay created by a gossip protocol. Due to high connectivity of many nodes in the physical network, using the physical connectivity graph for the management overlay is generally not suitable. The topological properties of the overlay topology and the relationship between overlay and underlying physical topologies are open issues for further work.

4.1.4. Failures

For our simulations, we assume that failure arrivals follow a Poisson process and that a failed node recovers after 30 s. We also assume a failure detection service is available to TCA–GAP, allowing a node to detect the failure of a neighbor in the network graph.

4.1.5. Other simulation parameters

We run the simulations with the following parameters unless stated otherwise. The choices for the particular values are based on the configuration and measurements on our testbed [7], internet measurements, and the need for a sufficient number of measurement events to obtain statistically significant simulation results.

- Maximum message rate: 4 msg/s per link.
- Protocol parameters: $k_1 = 0.9$, $k_2 = 0.85$.
- Processing overhead: 1 ms/message.
- Network delay across links of the graph: 20 ms.
- Length of a simulation run: 1500 s, with an initialization period of 5 s (tree construction starts at $t = 2$ s and threshold assignment begins at $t = 5$ s).
- Threshold values: T^{g+} is set at 1.05 times the average value of the aggregate and T^{g-} at the average value of the aggregate.

4.2. Protocol efficiency

We assess the efficiency of TCA–GAP by measuring the protocol overhead in two scenarios, one in which several threshold crossings occur, and a second scenario in which the threshold is not crossed.

In the first scenario, we run the protocol on the 654-node network graph where the local variables change according to Periodic UT trace. The simulation is run for 45 s and Fig. 6 shows the trace of the simulation.

Fig. 6 shows the change of the aggregate and the protocol overhead over time. During the simulation run, three threshold crossings occur: at around $t = 9$ s (upper threshold crossing), $t = 23.5$ s (lower threshold crossing) and $t = 39$ s (upper threshold crossing). Before each threshold crossing, e.g. between $t = 8$ s to $t = 10$ s, we observe a peak in protocol overhead. This can be explained by an increased level of threshold recomputation during those periods, as well as the transition of passive nodes to active state, which includes resetting of thresholds on subtrees. For each the three threshold crossings, we see a second peak in the protocol overhead, which relates to the root node becoming passive and thus distributing new threshold values to all nodes in a recursive manner.

The conclusions from this and other similar experiments we performed are that the protocol overhead is low whenever the aggregate is far from the threshold. Second, the protocol overhead is highest shortly before a threshold is crossed or for short period after the root node switches from active to passive state. Third, the protocol overhead of TCA–GAP during peak periods is comparable to the average overhead of a tree-based continuous aggregation protocol such as GAP (see Section 3.2). (For the above scenario, we measured the protocol overhead for GAP to be 1.7 msgs/s.)

In the second scenario, we study the effect of the distance of the aggregate from the threshold on the protocol overhead. To do this, we use the UT trace with the 654-node topology in a setting where no threshold crossings occur during the simulation runs. For each simulation run, we use different threshold values, so that the ratio of aggregate to the (upper) threshold ranges from 10% to 100%. Each simulation is run for 1500 s, and the average protocol overhead over this time is measured. Fig. 7 shows the result.

As can be seen from Fig. 7, the protocol generates almost no overhead as long as the ratio between the aggregate and the threshold is less than 40%, beyond which, the overhead increases almost exponentially until 90%. We also see that the increase in the protocol overhead from 90% to 100% is small, which we explain by the local hysteresis mechanism at the root node which causes the root node to be active in both cases. At 100%, the overhead is comparable to that of GAP which is measured to be 1.7 msg/s. (The 95% confidence intervals from these measurements are too small to be visible in the graph of Fig. 7.)

The scenario illustrates in a quantitative way that the protocol overhead decreases with the distance of the aggregate from the threshold. The relationship between

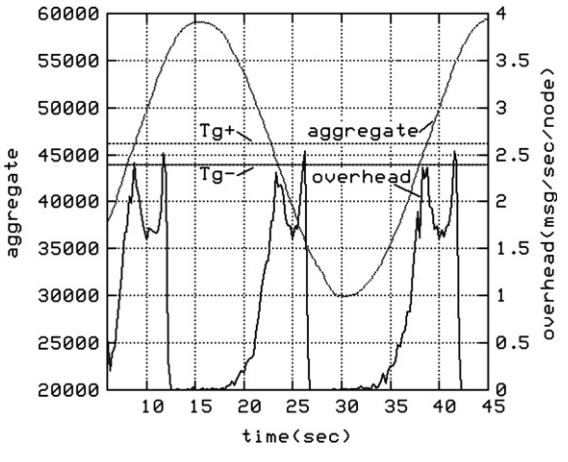

Fig. 6. TCA–GAP protocol overhead over time.

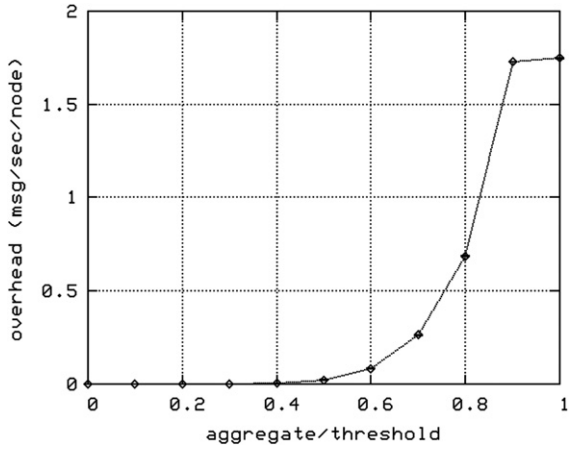

Fig. 7. Protocol overhead in function of average ratio of aggregate to threshold.

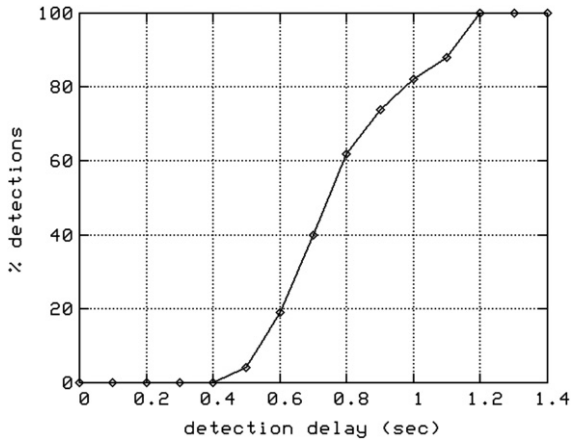

Fig. 8. Cumulative distribution of TCA detection delays.

the overhead and the distance is non-linear, and until a ratio of some 60%, the overhead is almost negligible.

4.3. Latency for threshold detection

In this scenario we study the delays for detecting a TCA by TCA–GAP. We simulate the protocol on the 654-nodes topology with the periodic UT trace. The protocol is run for 1500 s, resulting in 100 (50 upward and 50 downward) threshold crossings. The protocol did not exhibit false positives or false negatives during this simulation run. The resulting delay distribution is shown in Fig. 8.

The figure shows that, for this particular scenario, all threshold crossings are detected in between 500 ms and 1.75 s of their occurrence. The shape of this distribution depends on the dynamics of the local variables, the network size and the topology of the aggregation tree. Measurements from our testbed implementation have shown that a TCA alert is raised on rare occasions before the actual threshold crossing occurs [7]. Such rare events are much more likely to occur in small networks like our lab testbed (16 nodes) than on larger networks like in our simulation scenario.

4.4. Correctness

The correctness of TCA–GAP was assessed as part of an evaluation of a prototype implementation on our testbed [7,8]. There, we showed that the rate of occurrence of false positives and false negatives can be controlled by the maximum message rate and the control parameters k_1 and k_2. A higher message rate or lower values for k_1 and k_2 result in a lower probability of false positives or false negatives. This allows us to control the tradeoff between protocol overhead and correctness of TCA–GAP.

4.5. Scalability

We study the protocol in two scenarios, where we measure the protocol overhead and the TCA detection delay as a function of the network size.

Table 2
Topological properties of GoCast generated topologies used in the experiments

Size	Diameter	Average inter-node distance
82	5	2.9
164	5	3.6
327	6	3.9
654	7	4.3
1308	7	4.8
2616	8	5.3
5232	8	5.8

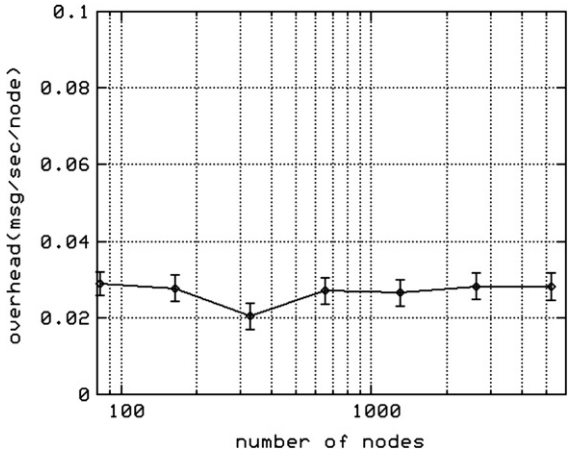

Fig. 9. Protocol overhead in function of network size.

In the first scenario, we use GoCast generated graphs with target connectivity of 5 for networks of size 82, 164, 327, 654, 1308, 2626 and 5232. Table 2 shows topological properties of these graphs. We use the UT trace to simulate the behavior of the local variables. For each topology, the threshold is set at twice the average of the aggregate during a run. The result is shown in Fig. 9.

Each point on the graph is the outcome of a simulation run. The figure suggests that the protocol overhead, measured in msg/s/node, for the specific settings of this scenario, does not depend on the network size. Note that the protocol limits the overhead per node by the maximum message rate, which is 4 msg/s/link in this scenario. Since our measurements suggest that the observed overhead is about two orders of magnitude lower than the imposed message rate limit, the maximum message rate does not explain the independence of the protocol overhead on the system size in this scenario.

From the above discussion, we conclude that TCA–GAP is scalable in system size with respect to the protocol overhead.

In the second scenario, we measure the average detection delay as a function of the network size. We use the topologies given in Table 2. The local variables are simulated using the periodic UT trace. The result is shown in Fig. 10.

Each point on the graph is the outcome of a simulation run. The figure suggests that the average TCA detection delay increases with the logarithm of the system size. Note

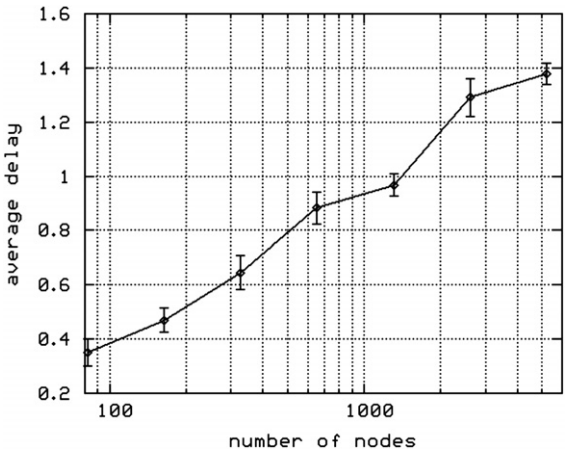

Fig. 10. Detection delay in function of network size.

that a tree-based aggregation protocol, such as GAP, where threshold detection would be a function of the root node, would exhibit asymptotically the same average detection delay as TCA–GAP, however, at a much higher overhead. Note also that for any tree-based aggregation protocol, threshold detection cannot be achieved below an average delay of O(log n), n being the system size. The asymptotic behavior of the average detection delay for TCA–GAP is dependent on the dynamics of the local variables and the choice of the variables k_1 and k_2. As we discuss in Section 5, it can be O(n) for a rare, worst case.

From the above discussion, we conclude that TCA–GAP is scalable in system size with respect to detection delay, for the right values of k_1 and k_2.

4.6. Robustness

We study the robustness property of TCA–GAP in three scenarios: first, we measure the protocol overhead as a function of the rate of node failures. Then, for a fixed failure rate, we measure the delay in detecting TCAs. Finally, we evaluate the accuracy of the protocol under node failures.

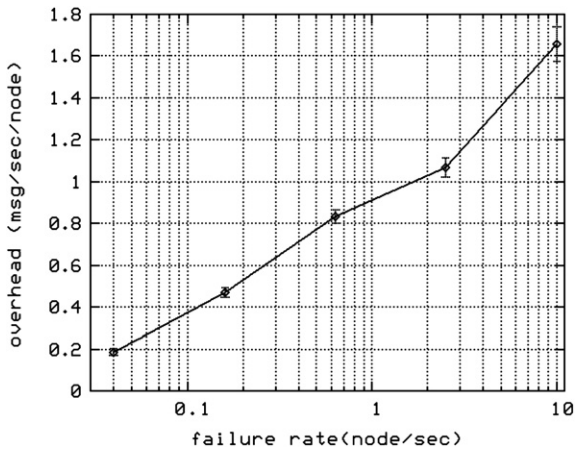

Fig. 11. Protocol overhead in function of failure rate.

For the first scenario, we use the default 654-node topology and we simulate local variables with the UT trace. The threshold is chosen five times the average value of the aggregate during the run. Then, for failure rates of 0.039, 0.156, 0.625, 2.5 and 10 failure/s/network, we measure the protocol overhead. The result is shown in Fig. 11.

Each point on the graph is the outcome of a simulation run. As can be seen from the figure, the protocol overhead seems to increase with the logarithm of the failure rate. At this point, we do not have a good explanation why this is the case. Note that the protocol overhead is limited by the maximum message rate, and the failure rate cannot be larger than 22 failures/s since failed nodes recover only after 30 s (see assumptions at the beginning of section).

An important observation is that failures introduce considerable overhead in the protocol, even for small failure rates. In Section 4.3, we report on results with the same topology and trace, which include a measurement with the same threshold as this scenario. There, the protocol overhead is virtually 0, which compares to a range of 0.2–1.7 msg/s/node for the failure rates considered in this experiment. We suspect that the high overhead occurs as many nodes may become active as a consequence of a change in the topology of the spanning tree, and the current version of the protocol does not attempt to switch active nodes into passive state, for example through threshold recomputation.

For the second scenario, we use the same setting as Section 4.3, i.e. we simulate the protocol on the 654-nodes topology with the periodic UT trace. We produce a simulation run with 1 failure/s/network and measure the detection delays. Fig. 12 shows the curve from this simulation run and that from Fig. 8.

As can be seen from the figure, the effect of failures is that the variance as well as the average of the detection delays increases. In this specific failure scenario, the distribution is spread from −0.5 s to 3.75 s, compared to from 500 ms to 1.5 s when no failures occur, and the average detection time increases from 0.78 s to 1.15 s.

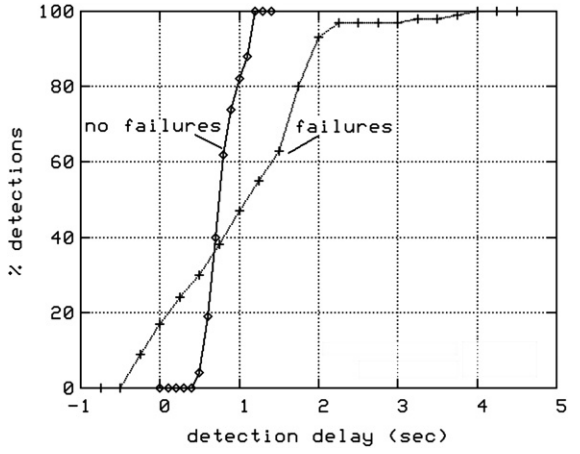

Fig. 12. Cumulative distribution of TCA detection delays for 1 failure/s/network and no failures.

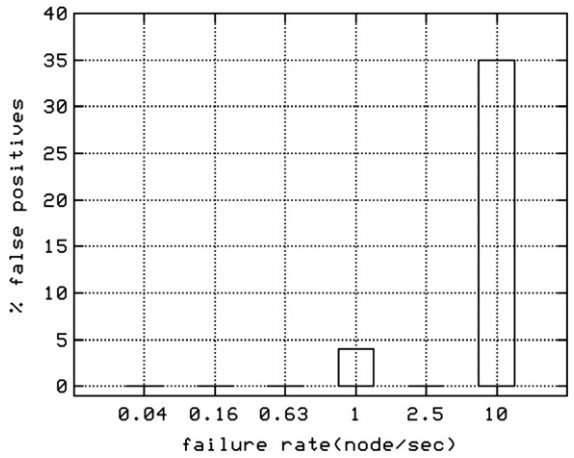

Fig. 13. False positives (as % of raised alarms) in function of failure rate.

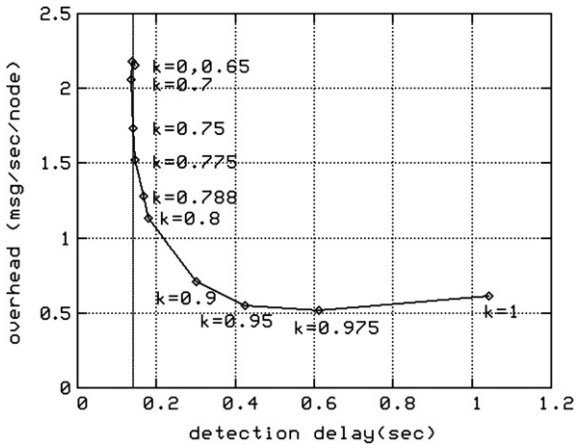

Fig. 14. k controls the tradeoff between protocol overhead and detection delay.

In the last scenario, we run simulations with failure rates of 0.039, 0.156, 0.625, 1, 2.5 and 10 failure/s/network, for the same setting as in the experiment above. For each simulation run, we assess the correctness of the protocol by determining the number of false negatives and false positives. The result is shown in Fig. 13.

The figure shows only the curve for false positives, as there were no false negatives measured during all simulation runs. As can be seen from the figure, the number of false positives tends to increase as the failure rate increases. We suspect that these false positives are caused by spikes in the estimate of the aggregate, which can occur during tree reconstruction in tree-based aggregation protocols.

4.7. Controllability

We study the controllability of TCA–GAP. Specifically, we are interested to which extent we can control the tradeoff between overhead and the quality of TCA detection. As control parameter, we use the k_1, which defines the upper threshold for the local hysteresis mechanism (see Section 3.3). (k_2 is chosen as 0.95^*k_1.) For simplicity reasons we refer to k_1 simply as k.

We measure the average protocol overhead and the average TCA detection delay as functions of k. For the scenario, we use the default parameters for the simulation (see Section 4.1) except for the maximum message rate, which is 10 msg/s instead of the default 4/s. We use the periodic UT trace to generate multiple threshold crossings. The scenario is run for values of k equal to 0, 0.65, 0.7, 0.75, 0.775, 0.788, 0.8, 0.9, 0.95, 0.975 and 1. The result is shown in Fig. 14.

The graph shows measurements for the full domain of $k \in [0.1]$. We observe that up to a certain value of k (around $k = 0.775$), the overhead reduces without an increase in detection delay. Beyond this value, the overhead generally decreases with an increase in the detection delay until k reaches 1.

For values of k between 0 and 0.667, we expected the same overhead and detection delays, because 0.667 is the ratio between the minimum of the aggregate during all simulation runs and the threshold. As a consequence, for values of k less than 0.667, all nodes are always active and the protocol executes GAP.

An interesting phenomenon is that up to a value of $k = k^* = 0.775$ the measurements do not show an increase in detection delay while there is a decrease in the overhead. We conclude that k can be increased above 0.667 to k^* such that a system running TCA–GAP exhibits the same detection delays as a system that runs GAP, although at a smaller overhead. We speculate that k^* depends on the topology and the dynamics of the local variables.

From this scenario we draw the following conclusions. First, k is an effective parameter for controlling the tradeoff between overhead and detection delay. Second, by allowing a small increase over the minimum detection delay, the overhead can be decreased significantly. Third, for values of k in the interval $[0, k^* < 1]$, TCA–GAP exhibits the same detection delays as a system that runs GAP, although a lower overhead for $k = k^*$.

5. Related work

Interest in the problem of detecting network-wide threshold crossings has built up in recent years. All approaches known to us use some types of thresholds or filters on individual nodes to reduce the protocol overhead. Most proposed protocols exhibit low overhead when the aggregate is far from the monitored threshold.

Depending on the degree of centralization of the proposed approaches, we classify the related work as either *weakly distributed* or *strongly distributed*, following the classification in [15]. Most published work has been done in the area of weakly distributed solutions. They assume a central coordinator that is responsible for computing local thresholds and filter parameters for all nodes. The coordinator also performs aggregation of the local variables and detection of network-wide threshold crossings. A common drawback of weakly distributed approaches is that the load on the coordinator increases linearly with the system size, which limits the scalability of the protocol.

In strongly distributed approaches such as the TCA–GAP protocol presented here, local thresholds or filters parameters are computed in a fully distributed manner. In particular, such approaches allow all nodes to interact only with their immediate neighbors on the network graph over which the protocol is running.

These solutions can further be divided into two groups. The first group is composed of protocols that support *linear aggregation* functions (e.g. SUM or COUNT) or aggregation functions that can be derived from such linear functions. The second group contains protocols that support aggregation functions beyond simple linear functions.

The earliest result known to us in the context of detecting network-wide threshold crossings is the work by [16], which presents several weakly distributed protocols for the SUM aggregation function. In an approach the authors call *simple-value*, a local threshold value of T/n where n is the number of nodes in the network, and T is the global threshold, is assigned to all nodes. Whenever the local variable crosses this threshold, the node sends a trap with the local variable to the management station. Periodically, the management station polls all nodes for the local variables, if it has received a trap during the last period. Then, it aggregates the local variables and determines whether the global threshold has been crossed. We believe that this scheme performs well for "small" networks, for which polling is feasible, where local variables are evenly distributed, and where the likelihood of a node exceeding its threshold is small. In the same paper, the authors propose a second approach, called *simple-rate*, which assumes an upper bound on the rate of change of local variables per unit time, which leads to a protocol with smaller overhead than simple periodic polling.

In [2], Keralapura et al. present algorithms that include network-wide threshold detection functionality. The algorithms give an estimate of the count of events in a network, whenever the count crosses some global threshold. The estimate should be within a maximum error of δ. The basic approach is to use local thresholds $t_{i,j} < t_{i,j+1}$, $j = 0,1,\ldots$ on all nodes i. Whenever N_i leaves the interval $[t_{i,j}, t_{i,j+1})$, node i contacts the coordinator. The authors propose two ways of setting the new thresholds. In the first approach, which they call *static thresholding*, all nodes are assigned a predetermined set of thresholds that do not change over time. In the second approach, which the authors call *adaptive thresholding*, the coordinator chooses new thresholds.

In contrast to TCA–GAP, this approach includes a central coordinator and hence is weakly distributed. Apart from that, the threshold assignment and recomputation in adaptive thresholding is similar to TCA–GAP in the sense that thresholds are assigned proportional to the contribution of the local variable to the aggregate, using a reactive scheme.

Huang et al. [1] present an approach for detecting network-wide threshold crossings for *non-linear* aggregation functions, which they also call *distributed triggers*. Similar to the approaches above, a coordinator continuously computes and distributes the available threshold space, i.e. the difference between the estimate of the aggregate and the threshold value, to all nodes, proportional to the variance of the local values. Each node receives a local threshold and sends updates to the coordinator whenever that is exceeded.

The authors illustrate their scheme through the use of principal component analysis (PCA) on link traffic measurements to identify anomalies in bandwidth consumption [17]. Taking time series of the link loads as input, the authors apply PCA, a method based on matrix algebra, to identify anomalous subspaces and to establish a metric through which anomalies are expressed and compared to a global threshold. To address the problem of non-linear aggregation functions, the authors use first-order approximation. By finding no false negatives in the data set they use for evaluation of their method, the authors conclude that their first-order approximation is sufficiently accurate.

While the authors demonstrate how threshold crossings with non-linear aggregation functions can be detected in an efficient way, their scheme still relies on a central coordinator, and they do not address the problem on how such a coordinator can be decentralized.

In [3], Sharfman et al. present an approach where threshold crossings are detected for a real-valued function that is computed over the sum of vector-valued local variables. The authors present a weakly distributed and a strongly distributed versions of their approach to detection of threshold crossing. In the first case, a coordinator sends its estimate of the aggregate vector to all nodes. Each node continuously evaluates a possible threshold crossing, by adjusting the estimate of the aggregate vector using its current local vector and evaluating the aggregation function within a ball centered at its adjusted estimate. If the node determines a possible crossing of the global threshold, then it sends a message to the coordinator, together with its current local value. The coordinator answers with an updated estimate of the global aggregate which is sent to a set of nodes. In the strongly distributed version of the scheme the authors propose, there is no central coordinator, but each node broadcasts its current local vector whenever it suspects a global threshold crossing.

We comment that the distributed scheme in this work is of a very limited scalability, as each node broadcasts its vector to all other nodes when the aggregate is close to the threshold. We expect that a different distributed approach would increase the scalability of the scheme, for instance an approach whereby the nodes are organized in a tree and each node acts as a coordinator for its children. Such a scheme would be quite similar to the one presented in this paper.

A strongly distributed solution to a problem closely related to the one tackled in this paper has been presented by Breitgand et al. in the context of estimating the size of a multicast group [18]. The authors address the problem of determining whether the group size is within an interval $[L, H]$ for which pricing is constant. They assume a synchronous network with a tree topology. Each leaf node i maintains a local variable $x_i(t)$ with $t = 0,1, 2, \ldots$, which represents the number of active multicast receivers at that node. The authors assume that the aggregate $\sum_i x_i(t)$ is piece-wise constant in the intervals $[t^*2\log^2 n, (t+1)^*2\log^2 n)$ where n is the number of nodes. They provide an algorithm, called HRMA, which detects when $\sum_i x_i(t)$ falls outside $[L, H]$. Initially, the root node computes local thresholds $\{l, h\}$ for all

leaf nodes and multicasts them. Then, every $2\log^2 n$ time units, the protocol determines whether the aggregate is within the interval $[L, H]$. First, each leaf node checks whether the local variable is outside of $[l, h]$. If it is, then the node sends a trap to its parent, who then polls all its leaf nodes for their current local values, computes the partial aggregate for all nodes i in its subtree, and checks whether the partial aggregate is within the interval $\left[\sum_i l_i, \sum_i h_i\right]$. If it is inside the interval, then the node stops the process. If it is outside, then it sends a trap to its parent, etc., in a recursive fashion. This process continues until a node stops it, or, the root node is reached, which then checks whether the global aggregate falls outside $[L, H]$, in which case the threshold is crossed. In such a case, the root node computes new local thresholds and multicasts them to all leaf nodes, and the algorithm starts monitoring the aggregate for the new interval. The authors discuss several modifications to HRMA which are designed to reduce the protocol overhead.

HRMA makes strong assumptions with respect to the network model, which is assumed to be synchronous and does not consider failures, as well as with respect to the sampling interval, which grows with the system size. Using these assumptions, HRMA can guarantee the detection of all threshold crossings. In contrast, TCA–GAP assumes an asynchronous network model, is robust to node and link failures, and makes no assumptions about the sampling interval. However, detection of all threshold crossings cannot be guaranteed, and false positives can occur, even if they are rare in our experience. Note though that TCA–GAP allows controlling the tradeoff between protocol overhead and the quality of TCA detection through the maximum message rate and the control parameters k_1 and k_2. From the perspective of applicability in a real network environment, the weak assumptions TCA–GAP makes with respect to the network model facilitates its implementation, whereas HRMA is much harder to realize.

We provide a comparison between HRMA and TCA–GAP with regard to TCA detection times and system sizes. First, we compare the asymptotic behavior of detection times with respect the system size. Second, we use a specific scenario to compare the detection times for comparable overhead.

Table 3 compares the asymptotic behavior of both protocols. In the case of HRMA, threshold detection is conducted every $2\log^2 n$ time units, which results in a maximum detection delay of $O(\log^2 n)$. A case of minimum detection time occurs when all local thresholds are crossed at the same time, causing the global threshold to be crossed. In this case, the threshold crossing is detected in $O(\log n)$ time, as the depth of the spanning tree is $O(\log n)$.

For TCA–GAP, if the local variable at the root node causes the global threshold to be crossed, then this is detected in $O(1)$ time. The maximum detection time of TCA–GAP depends on the value of k (which stands for the local hysteresis threshold k_1, see Section 4.7). For $k \in [0, k^*]$, the protocol has the same detection times as GAP with threshold detection at the root node, namely $O(\log n)$. For larger values of k, i.e. $k \in (k^*, 1]$, the maximum detection time is $O(n)$. The smaller detection time comes at the expense of a larger overhead, as we have shown in Section 4.7. (The above paragraph relates to TCA–GAP with Policy I. Policies II and III have other maximum detection times.)

From the discussion of the Table 3 we conclude that while the minimum detection time in TCA–GAP is asymptotically smaller than that of HRMA, the maximum detection time depends on the control parameter k. It can be chosen to be smaller or larger at the expense of protocol overhead.

Fig. 15 gives the average and maximum detection delays for TCA–GAP, taken from measurements for the scalability scenario in section 4.5. It also shows the average detection time for HRMA, which is analytically computed. For this purpose, we assume the average detection delay for HRMA to be $\log^2 n$ time units, which is based on the fact that threshold detection is performed every $2\log^2 n$ time units and on our assumption that a threshold crossing is equally likely to occur at any time. For computing the average detection delay, we further assume that the aggregation tree for HRMA has a branching factor 5. TCA–GAP is run with a message rate of 4 msg/s and a target connectivity of 5 for the network graph, which produces a maximum message rate of 20 msg/s a node has to process. A similar maximum overhead is achieved when HRMA is executed at 20 protocol cycles/s, i.e. HRMA progresses 20 time units per second.

As the average detection time for both protocols depends on the dynamics of the local variables and the topology of the aggregation tree, it is generally not possible to compare the complexities of the detection times for the average case. In the following, we give a comparison for a specific scenario in which we compare the protocols for a similar overhead. As a result, without specific assumptions regarding these entities, it is generally not possible to give

Table 3
Threshold detection times in function of system size n for HRMA and TCA–GAP

Protocol	Min	Max
HRMA	$O(\log n)$	$O(\log^2 n)$
TCA–GAP, $k \in [0,k^*]$	$O(1)$	$O(\log n)$
TCA–GAP, $k \in (k^*,1]$	$O(1)$	$O(n)$

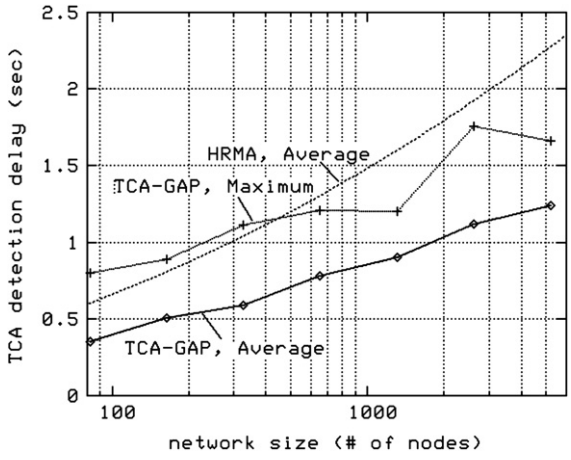

Fig. 15. TCA detection times: TCA–GAP vs. HRMA.

the complexity of the detection time in the average case. For this paper however, we perform a rough comparison as follows. For TCA–GAP we use the measurement of the average detection time from Section 4.5.

For HRMA, recall that threshold verification is done periodically every $2\log^2 n$ time units. Hence, if we ignore the time it takes the protocol to detect the threshold crossing once the verification has started, and if we assume that threshold crossing is equally likely to occur at any time in a given period, then the average time between the actual threshold crossing and the start of the threshold verification process is half of the length of the period which is $\log^2 n$ time units. Obviously, this is a lower bound to the average detection time as it does not take the time the threshold verification takes into account.

Fig. 15 shows the average and the maximum detection times of TCAs for the scalability scenario discussed in Section 4.5. The figure also gives the average detection time for HRMA as computed above, assuming for a round length of 50 ms and a spanning tree that is a full tree with a branching factor of 5. The round duration is chosen such that the two protocols have a comparable processing overhead, in the sense that they execute the same number of cycles/rounds in a given period. TCA–GAP is run with a maximum message rate of 4 msg/s and a target connectivity for the network graph of 5, resulting in a maximum load on a node of 20 msg/s, which is roughly equivalent to processing a message every 50 ms in HRMA.

The graph in Fig. 15 suggests that, for the chosen scenario and parameter range, TCA–GAP outperforms HRMA with respect to the average detection time. As we already observed in Section 4.5, the average detection time for TCA–GAP seems to increase with the logarithm of the system size, for the traces and the particular network graph used in this scenario.

6. Discussion and future work

The contribution of this paper is a detailed description and a comprehensive evaluation of TCA–GAP, a protocol for detecting threshold crossings of network-wide aggregates in a distributed way.

The design goals for the protocol outlined in Section 2 relate to *efficiency*, *quality of detection, scalability*, *robustness* and *controllability*. Regarding efficiency, the simulation results show that the protocol overhead is low whenever the aggregate is far from the threshold. For the series of experiments in Section 4.2, the protocol overhead is negligible for the runs where the average aggregate is below 60% of the threshold. The overhead becomes comparable to that of a tree-based aggregation protocol (such as GAP) when the aggregate is about to cross the threshold.

Regarding quality of detection, our results show that all threshold crossings are detected within 2 s for the scenario in Section 4.3, which uses a network topology with over 600 nodes. Our evaluation further shows that, for all scenarios included in this paper, all threshold crossings are indeed detected (i.e. no false negatives occurred). There is a chance of false positives in scenarios with a high rate of node failures, e.g. in Section 4.6, for failure rates of above 1 node failure/s in a 654-node network.

Our simulation studies suggest that TCA–GAP is scalable in system size. The results in Section 4.5 show that the protocol overhead is independent of the system size for the network sizes and scenario configurations considered. The observed average message rates of below 0.03 msg/s/node, for topologies with sizes from 82 to 5232 nodes, are well below the maximum possible rate of some 20 msg/s/node, which is imposed by the control parameter for the maximum message rate. In addition, the detection time of threshold crossings, for the scenarios in Section 4.5, grows with the logarithm of the system size. A tree-based aggregation protocol (such as GAP) has the same asymptotic behavior, although at a much higher overhead (1.7 msg/s/node compared to 0.025 msg/s/node for a network of 654 nodes in one of the scenarios presented).

Regarding robustness, the protocol overhead increases (approximately) with the logarithm of the failure rate in the network in the scenarios considered for our study (see Section 4.6). We also found that failures introduce considerable protocol overhead. With regards to detection times for threshold crossings, we observe that the average as well as the variance of the detection delays increases with the failure rate. No false negatives occurred and the number of false positives increased with the failure rate, for all simulation runs reported in this paper.

Finally, we demonstrated the controllability of our protocol. We identified the local hysteresis parameter k as an effective parameter for controlling the tradeoff between overhead and detection delay. Increasing k generally increases the detection delay while decreasing the overhead (see Section 4.7). Specifically, by allowing a small increase over the minimum possible detection delay, the overhead can be decreased significantly in the scenario we studied. We have argued that, for values of k in some interval $[0, k^* < 1]$, TCA–GAP exhibits the same detection delays as a system that runs GAP with threshold detection at the root node, with the lowest overhead for $k = k^*$.

We highlight two qualifications regarding the results of this paper. First, the traces used for the simulation runs are based on specific measurements from a university network (See Section 4.1). Had we taken measurements at a different time or from a different network, the simulation runs in this paper might have given different results. What gives us confidence in the overall conclusions we draw from the experiments in this paper is that other studies on an earlier version of TCA–GAP [6,7], which used a different set of traces, agree on a qualitative level with the measurement results reported in this paper. Note also that there are characteristics of TCA–GAP that do not depend on properties of the traces. These include the statements on threshold detection (Section 3.5) and the statements on the asymptotic behavior of threshold detection times (Section 5). Second, the current version of TCA–GAP is robust to crash failures only. Other types of Byzantine faults, which could be caused by software errors or malicious behavior, can not be detected and remedied by the protocol. This is an issue for further study.

The protocol presented in this paper is based on the concept of a TCA where alerts are instantaneously raised and cleared when the monitored variable crosses the relevant thresholds. Alternative conditions for raising alerts

have been proposed. For example, [1] suggests that an alert be raised based on for how long and/or by how much the variable stays above the threshold. A further alternative is raising an alert when the variable is within a specified interval around the threshold [1–3]. Supporting the above notions of TCAs can be achieved through straightforward extensions of TCA–GAP.

While we used the aggregation function SUM throughout the paper, TCA–GAP can be run with other aggregation functions, such as AVERAGE, MIN and MAX. For the case of AVERAGE, a straightforward solution whereby the problem is converted to that of monitoring a TCA over SUM is as follows. In addition to the monitored variable, which is aggregated using SUM, the network size is aggregated in the same way using COUNT. Then, for a given threshold T, an alert is raised whenever $\sum_i w_i(t) > T^*$, with $T^* = nT$. To run TCA–GAP with the MAX aggregation function, incremental aggregation is performed using MAX, and all local thresholds have the same value, which is equal to the global threshold. Threshold recomputation is never invoked in this case, since the violation of either threshold rule would imply that the local, as well as the global threshold is crossed. k_1 and k_2 are set to 1, since smaller values would increase overhead, while the threshold detection time would remain the same. (For the MIN aggregation function, incremental aggregation is performed using MIN; otherwise, the same discussion applies.) In terms of performance, we observed in experiments not reported in this paper that the aggregation functions MIN/MAX show significantly smaller overhead than SUM/AVERAGE in the same scenario. Contributing factors for this difference are that MIN/MAX does not incur any overhead regarding threshold recomputation and $k_1 = k_2 = 1$ for MIN/MAX.

The performance of TCA–GAP is influenced by the network graph over which the protocol runs. Specifically, the topology of the network graph influences the detection delays and the load distribution among the nodes. Our results from Section 4.5 and 4.7 clearly show that the detection time of TCAs depends on the depth of the aggregation tree. Suitable network graph topologies for TCA–GAP are those that enable a short detection delay and provide a balanced load among nodes. For the simulations reported in this paper, we used GoCast [14], a gossip protocol which has the target network connectivity as a configuration parameter, to create the network graph. In this context, the question arises, which is a suitable value for the connectivity of a GoCast graph, such that a short detection delay and a balanced load can be achieved. This aspect and the influence of the overlay topology on the protocol performance in general merits further investigation.

When considering the deployment of TCA–GAP in a production environment several issues need to be addressed. First, the protocol is robust to the failure of any node other than the root node. There are several practical options on how to mitigate a potential failure of the root, including running several instances of TCA–GAP with different root nodes. A related issue is the protocol execution in the case of a partitioned network, specifically in a partition without the root node.

As for future work, we plan on adapting the protocol for use in *highly dynamic networks* with continuous changes of topology or high failure rates. The measurements in Section 4.6 show that the protocol overhead can increase several orders of magnitude in a highly dynamic environment. As an alternative to tree-based aggregation protocols, we plan to consider gossip protocols for threshold detection. We expect gossip protocols to have a better performance in dynamic environments as they do not maintain a spanning tree.

A second topic we plan to address is the extension of TCA–GAP towards supporting *complex aggregation functions* and *complex triggers* for the purpose of anomaly detection and in network data mining. [1] provides a good example where a method called PCA is used to detect volume anomalies in network traffic. The fundamental challenge is whether such aggregation functions, or approximations thereof, can be efficiently computed in a distributed way inside the network.

Acknowledgement

The authors would like to thank Alex Clemm for his advice and fruitful discussions throughout this work. This work has been supported in part by a grant from Cisco Systems, the EC IST-EMANICS Network of Excellence (#26854), the EC 7th Framewotk 4WARD project, and the ACCESS Linnaeus Center at KTH.

References

[1] L. Huang, M. Garofalakis, J. Hellerstein, A. Joseph, N. Taft, Toward sophisticated detection with distributed triggers, in: Proceedings of the SIGCOMM 2006 Workshop on Mining Network Data (MineNet'06), Pisa, Italy, September 11–15, 2006.
[2] R. Keralapura, G. Cormode, J. Ramamirtham, Communication-efficient distributed monitoring of thresholded counts, in: Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data (SIGMOD'06), Chicago, IL, USA, June 27–29, 2006.
[3] I. Sharfman, A. Schuster, D. Keren, A geometric approach to monitoring threshold functions over distributed data streams, in: Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data (SIGMOD'06), Chicago, IL, USA, June 27–29, 2006.
[4] M. Dam, R. Stadler, A generic protocol for network state aggregation, in: Proceedings of the Radiovetenskap och Kommunikation (RVK 2005), Linköping, Sweden, June 14–16, 2005.
[5] S. Madden, M.J. Franklin, J.M. Hellerstein, W. Hong, TAG: A Tiny Aggregation service for ad-hoc sensor networks, in: Proceedings of the Fifth Symposium on Operating Systems Design and Implementation (OSDI'02), Boston, MA, USA, December 9–11, 2002.
[6] F. Wuhib, M. Dam, R. Stadler, A. Clemm. Decentralized computation of threshold crossing alerts, in: Proceedings of the 16th IEEE/IFIP Workshop on Distributed Systems: Operations and Management (DSOM 2005), Barcelona, Spain, October 24–26, 2005.
[7] F. Wuhib, R. Stadler, A. Clemm. Implementation and evaluation of a protocol for detecting network-wide threshold crossing alerts. Fourth IEEE/IFIP Workshop on End-to-End Monitoring Techniques and Services (E2EMON), Vancouver, Canada, April 3, 2006.
[8] F. Wuhib, R. Stadler, A. Clemm, Decentralized service level monitoring using network threshold crossing alerts, IEEE Communications Magazine 44 (10) (2006) 70–76.
[9] S. Dolev, A. Israeli, S. Moran, Self-stabilization of dynamic systems assuming only read/write atomicity, in: Proceedings of the Ninth Annual ACM Symposium on Principles of Distributed Computing (PODC'90), Quebec City, Quebec, Canada, August 22–24, 1990.
[10] IEEE. ANSI/IEEE Std 802.1D, 1998 Edition, IEEE, 1998.
[11] K.S. Lim, R. Stadler, Weaver – realizing a scalable management paradigm on commodity routers, in: Proceedings of the Eighth IFIP/IEEE International Symposium on Integrated Network Management (IM 2003), Colorado Springs, Colorado, USA, March 24–28, 2003.
[12] K.S. Lim, R. Stadler. SIMPSON – a simple pattern simulator for networks, <http://www.s3.kth.se/lcn/software/simpson.shtml>, February 2007.

[13] University of Twente – Traffic Measurement Data Repository, <http://m2c-a.cs.utwente.nl/>, February 2007.
[14] C. Tang, C. Ward, GoCast: Gossip-enhanced overlay multicast for fast and dependable group communication, in: Proceedings of the International Conference on Dependable Systems and Networks (DSN'05), Yokohama, Japan, June 28–July 1, 2005.
[15] J.P. Martin-Flatin, S. Znaty, A simple typology of distributed network management paradigms, in: Proceedings of the Eighth IFIP/IEEE International Workshop on Distributed Systems: Operations and Management (DSOM'97), Sydney, Australia, October 1997.
[16] M. Dilman, D. Raz, Efficient reactive monitoring, IEEE Journal on Selected Areas in Communications (JSAC) Special Issue on Recent Advances in Network Management 20 (4) (2001) 668–676.
[17] A. Lakhina, M. Crovella, C. Diot, Diagnosing network-wide traffic anomalies, ACM SIGCOMM Computer Communication Review 34 (4) (2004) 219–230.
[18] D. Breitgand, D. Dolev, D. Raz, Accounting mechanism for membership size-dependent pricing of multicast traffic, in: Proceedings of the Fifth International Workshop on Networked Group Communications (NGC 2003), Munich, Germany, September 16–19, 2003.
[19] N. Spring, R. Mahajan, D. Wetherall, T. Anderson, Measuring ISP topologies with rocket fuel, IEEE/ACM Transactions on Network 12 (1) (2004) 2–16.

Fetahi Wuhib (www.ee.kth.se/~fzwuhib) is a Ph.D., candidate at the Royal Institute of Technology (KTH), Stockholm, Sweden. He received his B.Sc. degree in Electrical Engineering from Addis Ababa University, Ethiopia, his M.Sc. degree in Internet working in 2005 and his Licentiate of Technology degree in Telecommunications from KTH in 2007. His current research focuses on protocols for decentralized monitoring and self-management.

Mads Dam (www.csc.kth.se/~mfd) is Associate Professor at the School of Computer Science and Communication, the Royal Institute of Technology (KTH), Sweden, since 1998, where he is a member of the theory group. He received his B.Sc. in Information Technology in 1983, and his M.Sc. in Computer Engineering in 1985 from Aalborg University, Denmark. In 1990 he received his Ph.D., in computer science from the University of Edinburgh, UK, where he remained as a post-doctoral researcher until 1992 when he moved to the Swedish Institute of Computer Science (SICS) to work as a research scientist. During the period 1994–2003 he headed the Formal Design Techniques Laboratory at SICS. During his research career, he has made significant contributions in areas such as modal and temporal logics, process algebra and mobile processes, program verification and program specification, and computer security. He is a member of NordSec steering committee and is a frequent member of program committees in the field of programming languages and security. His current research focuses partly on network management and autonomic computing, and partly on language-based approaches to computer security.

Rolf Stadler (www.ee.kth.se/~stadler) is a professor at the School of Electrical Engineering with the Royal Institute of Technology (KTH) in Stockholm, Sweden, since 2001, where he leads the Network Management Group. He received an M.Sc. degree in Mathematics in 1984 and a Ph.D., in Computer Science in 1990 from the University of Zurich, Switzerland. In 1991 he was a post-doctoral researcher at the IBM Zurich Research Laboratory. 1992–1994 he was a visiting scholar at the Center for Telecommunications Research at Columbia University, which he joined in 1994 as a research scientist. 1998–1999 he was a visiting professor at ETH Zurich. Over the last 10 years, he has been instrumental in the network management research community and has been program co-chair for premier IEEE conferences in the field, including DSOM'99, NOMS'02, and DSOM'07. He currently serves on the editorial board of IEEE Transactions on Network and Service Management (TNSM). His current research interests include scalable networks and systems, autonomous computing and self-management.

His research is supported by Swedish funding agencies (SSF, VINNOVA STINT), the European Commission, and direct grants from industry (Cisco Systems, IBM Research).

He is co-director of the Laboratory for Communication Networks (LCN) at KTH, together with Prof. Gunnar Karlsson. He is affiliated with the ACCESS Linneus Center at KTH, Wireless@KTH, and the Center for Networked System (CNS) at the Swedish Institute for Computer Science (SICS).