A GENERIC PROTOCOL FOR NETWORK STATE AGGREGATION

Mads Dam and Rolf Stadler

Dept. of Microelectronics and Information Technology, KTH, Stockholm, {stadler,mfd}@imit.kth.se

ABSTRACT

Aggregation functions, which compute global parameters, such as the sum, minimum or average of local device variables, are needed for many network monitoring and management tasks. As networks grow larger and become more dynamic, it is crucial to compute these functions in a scalable and robust manner. To this end, we have developed GAP (Generic Aggregation Protocol), a novel protocol that computes aggregates of device variables for network management purposes. GAP supports continuous estimation of aggregates in a network where local state variables and the network graph may change. Aggregates are computed in a decentralized way using an aggregation tree. We have performed a functional evaluation of GAP in a simulation environment and have identified configuration choices that potentially allow us to control the performance characteristics of the protocol.

1 INTRODUCTION

Network management systems collect state variables from network devices and process them to perform global control actions in accordance with management objectives. Since management is a global operation, management applications generally involve monitoring global parameters that are functions of local variables on devices, rather than monitoring individual variables [3]. For example, one is interested in the average load of UDP traffic across all network links, rather than in the load on individual links. Similarly, one may seek the sum of the SYN packets entering an organizational network through various gateways, rather than the number of packets on a each gateway, or the total number of service requests directed to a distributed web site, rather than the number of requests each server processes, etc.

Global parameters used by management applications are often aggregation functions, such as sum, max and average of device-level counters. In addition, more complex aggregation functions, including SQL-style queries have recently been proposed (cf. [10, 11]).

In traditional management approaches, the aggregation of local variables from different devices is performed in a centralized way, whereby an application running on a management station retrieves state variables from agents in network devices and performs the aggregation on the management station. For small and static networks, such approaches have proved efficient and effective. However, in the case of large networks and dynamic network environments, including sensor and mobile networks, these approaches have well-known drawbacks with respect to scalability and fault tolerance. A centralized architecture can lead to a large amount of management traffic exchanged between the management station and the agents, a high processing load on the management station, and long execution times for management operations, since all these metrics generally grow linearly with the number of network nodes. Further, the management station is a single point of failure in a centralized architecture, which makes engineering a robust system difficult.

The focus of this paper is on a distributed scheme for computing aggregation functions for network management purposes in a scalable and robust manner. Specifically, we present GAP (Generic Aggregation Protocol), a novel protocol that allows a management station to continuously receive an estimate of the current state aggregate $\Pi_n w_n(t)$, where $w_n(t)$ is the local value associated with node n at time t, and where multiplication represents aggregation. We generally use the generic term weight to represent a state variable on a network node. Weights can be router loads, sensor data, or traffic statistics. The weights are aggregated by an aggregation function $*$, which is assumed to be commutative, associative, and possesses an identity element 1.

We assume that each network device executes the protocol in a management process, either internally or on an external, associated device. These management processes communicate via a network overlay. We refer to this overlay as the network graph. A node in this graph is a network device together with its management process.

Our design goals for GAP are as follows:

- The convergence time, i.e., the time between a weight change on a node and the time this change is reflected in the aggregate on the management station, must be short.
- GAP must be robust with respect to node failures and recoveries. This means that a weight of a failed node will be removed from the aggregate until the node recovers. Also, the weights of new nodes that join the network, must be reflected in the aggregate, all within tolerable delays and error margins.
- GAP must provide efficient and scalable operation. This means that the load on the nodes must be balanced and the number of messages exchanged in the overlay must be small compared to a centralized aggregation protocol.

GAP is based on a distributed algorithm that builds and maintains a BFS tree on a network graph, following the approach of Dolev, Israeli and Moran [4]. GAP uses this tree to propagate aggregates back to the root, which is the start node of the protocol. This backpropagation may introduce spurious errors, but it can be shown that for static networks (no failures, no node creations) the worst case convergence time is no worse than the $2ld$ response time for the wave

propagation based solution where l is the maximal link delay and d the network diameter.

In this paper, we provide a description of GAP (Section 3) and briefly report on a functional evaluation in a simulation environment (Section 4). A more complete functional evaluation will be included in an extension of this paper. An evaluation of performance of GAP in terms of convergence times and message overhead is part of our current work (see Section 5).

2 RELATED WORK

The problem of distributed aggregation has received quite some attention recently [13]. In the domain of sensor networks, Madden et al. [11] use a tree topology maintenance approach closely related to ours. Their work is geared towards the sensor network setting, and in particular their protocol mixes in failure detection and MAC layer concerns, such as link quality, which in our approach is kept separate from the basic aggregation algorithm.

In [5] various difficulties involved in the distributed computation of aggregates are discussed, and a solution is proposed which uses hierarchical groups and gossiping. This work, however, focuses on one-shot distributed function evaluation in contrast to continuous monitoring. Gossiping has been considered as an approach to distributed aggregation by other authors as well. For instance, [6] describes a proactive scheme which "pushes" an aggregate to all nodes in an overlay network using an anti-entropy scheme. This work, however, assumes properties of the overlay network (either fully connected or a connected unbiased random topology) which makes its applicability as a general solution to the aggregation problem unclear. Other work dependent on particular forms of overlays include [7], which uses a DHT-based p2p overlay as a basis for constructing aggregation trees.

In our earlier work [8, 9], we have introduced a protocol for distributed state aggregation based on an echo algorithm. This protocol supports distributed polling of an aggregate state variable, which is a one-shot operation. GAP on the other hand, supports continuous estimation of an aggregate, while local state variables and the network graph may change. Both protocols use a spanning tree to incrementally aggregate the local variables.

The paper [3] proposes two algorithms that produce an event when the threshold of an aggregate function has been reached. Their solution is based on a centralized architecture. The goal of the algorithms is to reduce the monitoring communication overhead. The main difference to our work is the objective: While GAP produces a continuous estimate of a global aggregate function, their algorithms produce events, when a given threshold of such a function has been reached.

A particular difficulty which our works shares with most other approaches based on aggregation trees, is that link and node utilization increases with decreasing distance to the root. In this paper this is addressed by imposing a fixed upper bound on the frequency with which update messages can be back-propagated. More sophisticated approaches have been explored by other authors, for instance to impose local thresholds for backpropagation while still aiming to provide global error guarantees [2]).

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

Figure 1. Sample node table

3 THE GAP PROTOCOL

Essentially, the GAP protocol is a modified version of the BFS algorithm of Dolev, Israeli, and Moran [4]. The protocol of [4] works in coarsely synchronized rounds where each node exchanges its beliefs concerning the minimum distance to the root with all of its neightbours and then updates its belief accordingly. Each node also maintains a pointer to its parent, through which the BFS tree is represented.

The GAP protocol amends this basic algorithm in a number of ways:

1. The protocol uses message passing instead of shared registers.
2. Each node needs to maintain information about its children in the BFS tree, in order to correctly compute aggregates, and it performs the actual aggregation.
3. GAP is event-driven. This reduces traffic overhead at the expense of self-stabilisation, and it introduces some twists to ensure that topology changes are properly handled.

3.1 Data Structures

In GAP, each node n maintains a *neighbourhood table* T such as the one pictured in fig. 1 containing an entry for itself and each of its neighbours. In stable state the table will contain an entry for each live neighbour containing its identity, its status vis-a-vis the current node (self, parent, child, or peer), its level in the BFS tree (i.e. distance to root) as a non-negative integer, and its aggregate weight (i.e. the aggregate weight of the spanning tree rooted in that particular node). The exception is *self*. In that case the weight field will contain only the weight of the local node.

Initially, the neighbourhood table of all nodes n except the root contains a single entry $(n, \textit{self}, l_0, w_0)$ where l_0 and w_0 is some initial level, resp. weight. The initial level must be a non-negative integer. The initial neighbourhood table of the root contains in addition the entry

$$(n_{\text{root}}, \textit{parent}, -1, w_{\text{root}})$$

where n_{root} is a "virtual root" node id used to receive output and w_{root} is arbitrary. This virtual root convention ensures that the same code can be used for the root as for

other nodes, unlike [4] where the root is hardwired in order to ensure self-stabilization.

3.2 The Execution Model

The protocol executes using asynchronous message passing. The execution model assumes a set of underlying services including failure detection and neighbour discovery, local weight update, message delivery, and timeout, delivering their output to the process inqueue as messages of the form $(\texttt{tag}, Arg_1, ..., Arg_n)$. The following five message types are considered:

- $(\texttt{fail}, n)$ is delivered upon detecting the failure of node n.
- $(\texttt{new}, n)$ reports detection of a new neighbour n. At time of initialisation, the list of known neighbours is empty, so the first thing done by the protocol after initialisation will include reporting the initial neighbours.
- $(\texttt{update}, n, w, l, p)$ is the main message, called an *update vector*, exchanged between neighbours. This message tells the receiving node that the BFS tree rooted in sending node n has aggregate weight w and that n has the level and parent specified. This message is computed in the obvious way from n's neighbourhood table using the operation *upd_vector*(T). Observe that the parent field of the update vector is defined only when n's neighbourhood table has more than one entry.
- $(\texttt{weight}, w)$ is delivered as a result of sampling the local weight. The frequency and precision with which this takes place is not further specified.
- $(\texttt{timeout})$ is delivered upon a timeout.

3.3 Ancillary Functions

The algorithm uses the following ancillary functions:

- $\texttt{newentry}(n)$ returns n and creates a table entry

  $$T(n) = (n, s, l_0, w_0)$$

  where $s =$ *peer*, and l_0 and w_0 are suitable default values. If the row $T(n)$ already exists, no change is performed.
- $\texttt{removeentry}(n)$ removes the row $T(n)$, if it exists.
- $\texttt{updateentry}(n, w, l, p)$ assigns w and l to the corresponding fields of $T(n)$, if they exist, otherwise the row is first created. If $p =$ *self*() then $T(n)$.*Status* becomes *child*. This reflects the situation where n says that *self*() is parent. Otherwise, if $T(n)$.*Status* $=$ *child* then $T(n)$.*Status* becomes *peer*.
- $\texttt{level}(n)$ returns the level of n in T, if it exists, otherwise the return value is undefined.
- $\texttt{parent}()$ returns the node id n such that $T(Node)$.*Status* $=$ *parent*. If no such node id exists, a special value `nd` representing the undefined node is returned.
- $\texttt{send}(n, v)$ sends the update vector v to the node n.
- $\texttt{broadcast}(v)$ sends v to all known neighbours, not necessarily in atomic fashion.

3.4 The Algorithm

The main loop of the algorithm is given in pseudocode in fig. 2. Each loop iteration consists of three phases:

1. Get the next message and update the table accordingly.
2. Update the neighbourhood table to take the newly received information into account (the operation `restoreTableInvariant`).
3. Notify neighbours of state changes as necessary. In particular, when a new node has been registered, the update vector must be sent to it to establish connection, and when the update vector is seen to have changed and sufficient time has lapsed, the new update vector is broadcast to the known neighbours.

Much of the semantics of the GAP protocol is embodied in the operation `restoreTableInvariant`. Part of the tasks of this operation is to ensure a set of basic integrity properties of the neighbourhood table such as:

- Each node is associated with at most one row.
- Exactly one row has status *self*, and the node of that row is *self*().
- If the table has more than one entry it has a parent.
- The parent has minimal level among all entries in the neighbourhood table.
- The level of the parent is one less than the level of self.

In addition an implementation of the `restoreTableInvariant` operation may implement some policy which serves to optimize protocol behaviour in some respect such as:

- Optimization of convergence time, or
- Minimization of overshoot/undershoot after faults, or during initialisation.

Example policies are:

- Conservative: Once the parenthood relation changes, all information stored in the neighbourhood table except neighbour identities and levels and status of parent and self becomes, in principle, unreliable and could be forgotten (assigned the undefined value). This policy will ensure that the aggregate is, in some reasonable sense, always a lower approximation of the "real" value.
- Cache-like: All information stored in the neighborhood table, except information concerning *self* or status of parent, is left unchanged. This appears to be a natural default policy, as it seems to provide a good compromise between overshoot/undershoot and convergence time.

```
proc gap() =
  ... initialize data structures and services ...
  Timeout = 0 ;
  New = null ;
  Vector = updatevector();
  ... main loop ...
  while true do
  receive
    {new,From} =>
      NewNode = newentry(From) ;
  | {fail,From} =>
      removeentry(From)
  | {update,From,Weight,Level,Parent} =>
      updateentry(From,Weight,Level,Parent)
  | {updatelocal,Weight} =>
      updateentry(self(),Weight,level(self()),parent())
  | {timeout} => Timeout = 1
  end ;
  restoreTableInvariant() ;
  NewVector = updatevector();
  if NewNode != null
    {send(NewNode,NewVector); NewNode = null} ;
  if NewVector != Vector && Timeout
    { broadcast(NewVector); Vector = NewVector; Timeout = 0 }
od ;
```

Figure 2. Main loop of algorithm.

- Greedy or adaptive policies: Other policies can be envisaged such as policies which attempt to predict changes in neighbour status, such as proactively changing the status of a peer once its level has seen to be 2 or more greater than *self*'s. It is also possible to adapt the tree topology to MAC layer information such as link quality, as is done in [11].

4 EXPERIMENTAL EVALUATION

We have implemented GAP on SIMPSON, a network simulator that allows to study protocols on network graphs [1]. To date, we have performed simulation studies that allow a functional evaluation of GAP in different scenarios. We have studied the behavior of GAP during its initialization phase, during periods of random weight changes on nodes and during periods of random node failures and recoveries.

In the following, we show the results from two simulation runs using the topology of Tiscali, an ISP in the U.S., which has 506 nodes and 750 links. This topology was obtained from results of the Rocketfuel project [12]. We chose the overlay topology to be identical to the physical network topology. The root node was selected near the center of the network.

For the simulation experiments, the message size was chosen to be 1KB, the link delay 4ms, the protocol overhead 1ms, and the execution delay 1ms. We ran GAP with a cache-like policy for the operation `restoreTableInvariant` (Section 3).

Scenario 1 (Weight change): All weights are initialized with 10. At t=0, GAP starts initializing in the network graph. At t=0.1, the root node starts invoking the aggregation function SUM on GAP. Between 1 sec and 3 sec, 50 randomly selected nodes change their weights to random values that are uniformly distributed within [0,100].

Scenario 2 (Node failures and recoveries): All weights are initialized with 10. At t=0, GAP starts initializing in the network graph. At t=0.1, the root node starts invoking the aggregation function SUM on GAP. During the first 2 seconds, nodes randomly fail at times that follow a random exponential distribution with mean 0.01 sec. The recovery times follow the same distribution.

Figures 3 and 4 show measurements from the simulation runs. In both figures, the curve labeled actual represents the maximal achievable values for the aggregation function SUM, i.e., the values an ideal system with infinite resources and no communication delay would produce. The second curve, labeled measured, gives the values computed by GAP. The difference between the curves indicates the estimation error.

Figure 3 shows how GAP eventually converges towards the maximal achievable values, once no further weight changes occur, i.e., after 3 sec simulation time. The same behavior can be observed in Figure 4, which shows how GAP operates in a scenario with node failures and recoveries.

Figure 4 illustrates a striking property of the protocol in form of large undershoots and overshoots of some estimated values when compared to the maximum achievable values. We explain this by the fact that node failures and recoveries involve reconfiguration of the aggregation tree. During the transient phase of reconfiguration, the estimates for the aggregation function will be wrong. A close inspection shows that estimates during a

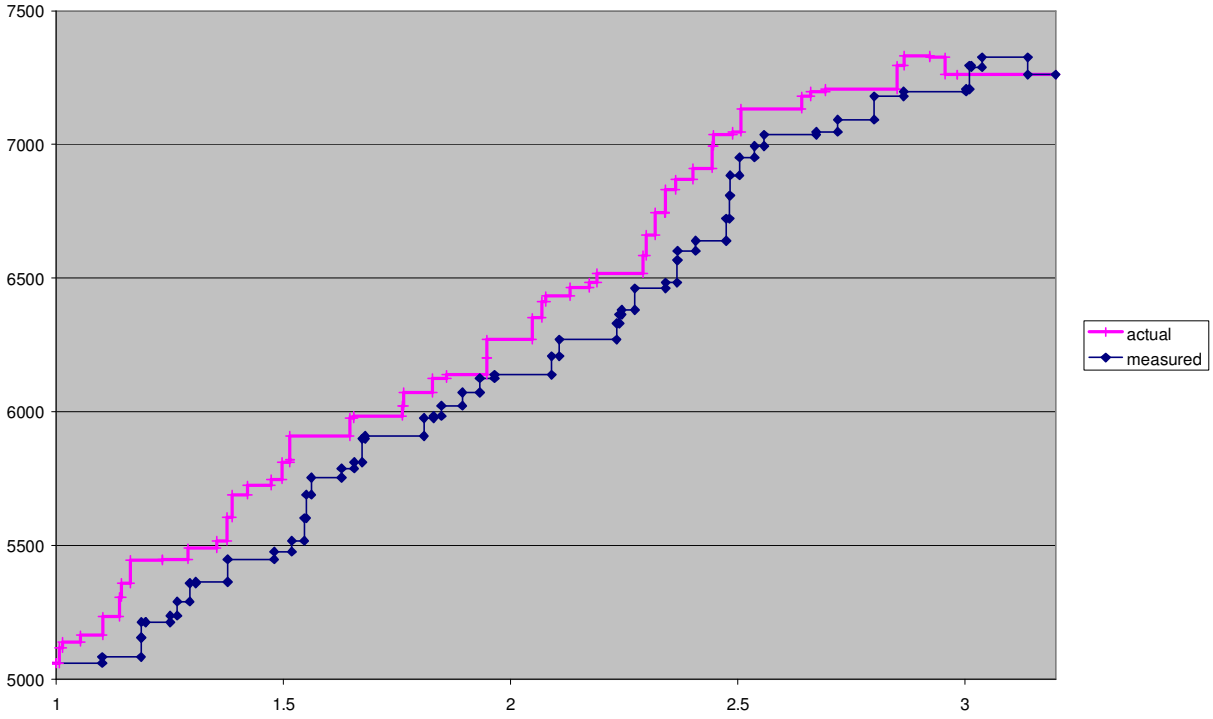

Figure 3. Actual and estimated (measured) values vs. time for the aggregation function SUM in scenario 1.

transient phase depend on the specific policy for the operation `restoreTableInvariant`. We conclude from this and other experiments that a different policy needs to be found that minimizes such spikes in the estimation error. A second insight Figure 4 provides is that node failures during the initialization phase of GAP significantly extend the duration of this phase.

The two simulation scenarios described above and many more not reported in this paper provide experimental proof that the functional behavior of GAP is as intended: The protocol initializes correctly on various network topologies and provides estimates that converge in a short period of time towards exact values of an aggregation function, once no further changes in local weights occur. It is robust with respect to node failures and recoveries in that the estimated values converge towards the exact values, once no further failures and recoveries occur in the network.

5 DISCUSSION

GAP, as defined in this paper, is a generic protocol and thus not bound to any particular aggregation function. In actual implementations, the aggregation function can be chosen at runtime and is then distributed along the aggregation tree, from the root towards the leaf nodes. Such systems have been independently proposed by Lim and Stadler [8] for Internet management and Madden et al. in the context of sensor networks [11].

While this paper describes the design of GAP and its functional evaluation, our current work focuses on evaluating the performance characteristics of GAP, specifically aspects of scalability.

An important issue is to identify configuration choices that can be used to control the performance of GAP. For instance, the choice of the overlay topology clearly affects the performance characteristics of the protocol. While, for instance, a high degree of connectivity in the overlay reduces the depth of the aggregation tree and thus can shorten the delay for a weight change to be reported to the root, the same configuration can lead to overload of nodes close to the root.

A further avenue of investigation is directed towards identifying trigger functions for weight subscription and update dissemination in GAP. Triggers can be based on timers or thresholds related to weights or partial aggregates. One of the interesting problems in this context is: given a maximum tolerable estimation error, identify trigger functions for all network nodes, such that the number of messages and the load on the nodes are minimized. Roussopoulos et al. describe a possible approach in [2].

Acknowledgements This work has been supported by VINNOVA under the project “Policy-Based Network Management”. The functional evaluation of GAP has been performed by Fetahi Wuhib, an M.Sc. candidate at IMIT, KTH.

REFERENCES

[1] SIMPSON — a SIMple Pattern Simulator fOr Networks.

[2] N. Roussopoulos A. Deligiannakis, Y. Kotidis. Hierarchical in-network data aggregation with quality guarantees. In *Proc. 9th International Conference on Extending Database Technology (EDBT)*, March 2004.

[3] M. Dilman and D. Raz. Efficient reactive monitoring. *IEEE Journal on Selected Areas in Communications (JSAC)*, 20(4), 2002.

[4] S. Dolev, A. Israeli, and S. Moran. Self-stabilization of dynamic systems assuming only read/write atomicity. *Distributed Computing*, 7:3–16, 1993.

[5] I. Gupta, R. van Renesse, and K. Birman. Scalable fault-tolerant aggregation in large process groups. In *Proc. Conf.*

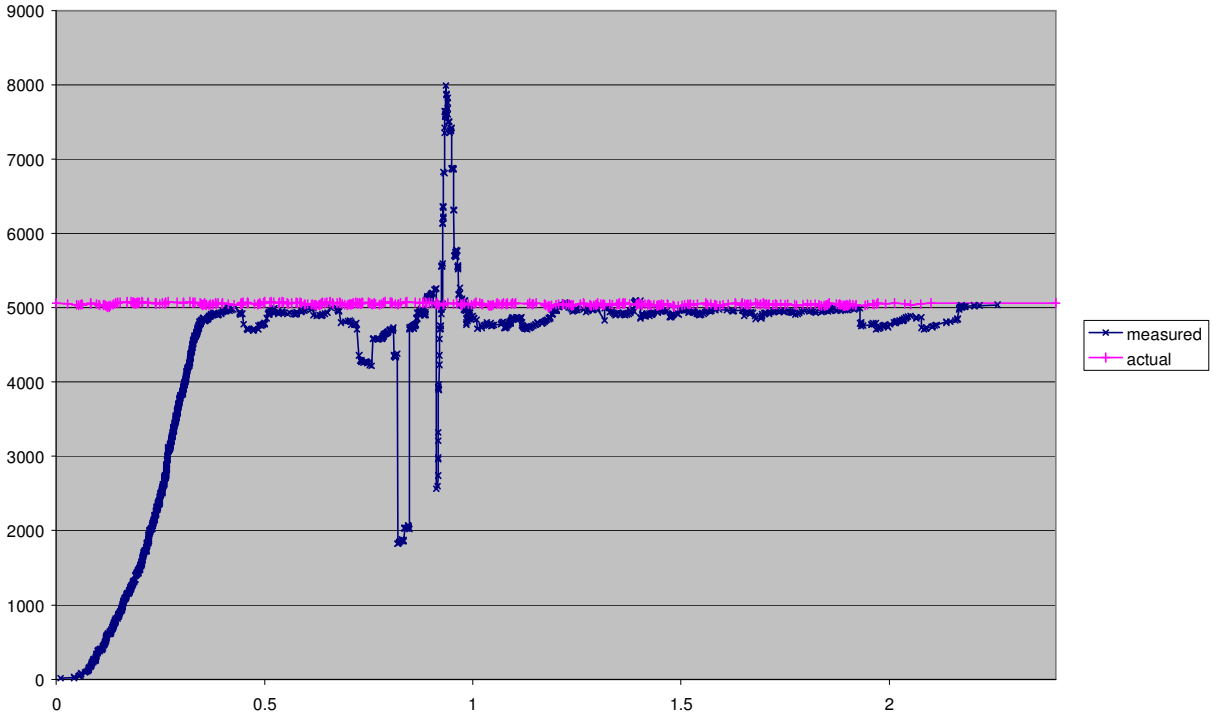

Figure 4. Actual and estimated (measured) values vs. time for the aggregation function SUM in scenario 2.

on Dependable Systems and Networks., pages 433–442, 2001.

[6] Márk Jelasity and Maarten van Steen. Large-scale newscast computing on the Internet, October 2002.

[7] Ji Li and Dah-Yoh Lim. A robust aggregation tree on distributed hash tables. In Vineet Sinha, Jacob Eisenstein, and Tefvik Metin Sezgin, editors, *Proceeings of the 2004 Student Oxygen Workshop*, September 2004.

[8] K. S. Lim and R. Stadler. A navigation pattern for scalable internet management. In *Proc. 7th IFIP/IEEE Int. Symp. on Integrated Network Management (IM 2001)*, 2001.

[9] K. S. Lim and R. Stadler. Weaver — realizing a scalable management paradigm on commodity routers. In *Proc. 8th IFIP/IEEE Int. Symp. on Integrated Network Management (IM 2003)*, 2003.

[10] K. S. Lim and R. Stadler. Real-time views of network traffic using decentralized management. In *Proc. 9th IFIP/IEEE Int. Symp. on Integrated Network Management (IM 2005)*, 2005.

[11] S. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong. TAG: A Tiny AGgretation service for ad-hoc sensor networks. In *Proc. 5th Symposium on Operating Systems Design and Implementation*, pages 131–146, 2002.

[12] N. Spring, R. Mahajan, and D. Wetherall. Measuring ISP topologies with Rocketfuel. In *Proc. ACM/SIGCOMM*, 2002.

[13] R. van Renesse. The importance of aggregation. In *In (A. Schiper, A.A. Shvatsman, H. Weatherspoon, and B. Y. Zhao, eds.), Future Directions in Distributed Computing, Lecture Notes in Computer Science*, volume 2584, pages 87–92. Springer-Verlag, 2003.