Protocols for Distributed Management

Rolf Stadler
KTH Royal Institute of Technology, Sweden
stadler@kth.se


May 31, 2012

Abstract

We survey approaches to distributed management and highlight an architecture that is especially suited for distributed management in a large network. We then discuss in detail two fundamental classes of protocols that execute within such an architecture. The first is the class of Echo protocols, which can be used for distributed polling, global state estimation, resource discovery, and distributed configuration. The second class is that of GAP protocols, whose main application is continuous real-time monitoring. Both classes are based on distributed trees, which are created during the execution of the protocols. Furthermore, both protocols perform in-network aggregation of the results from local operations on network elements. When presenting the protocols, we discuss their underlying distributed algorithms, their performance properties, such as overhead and execution times, and possible extensions for operational use. We limit the discussion to a single administrative domain.

1 Introduction

Traditional network and systems management follows an interaction paradigm with two distinguishing features. First, most management operations are performed on a per-device basis. When monitoring a network, for example, a management station typically polls devices in its domain periodically for the values of local variables, which are then processed on a management station. The same pattern of per-device interaction characterizes virtually all management tasks, including fault, configuration, accounting, performance and security management (often abbreviated as FCAPS). SNMP is probably the best-known protocol that supports this interaction paradigm [13]. Second, traditional management systems generally assume low-level interfaces in network elements. They provide access to SNMP managed objects, NetFlow aggregates (see, e.g., [3]) or CLI (command-line-interface) commands. As a result, the ‘management intelligence’ of the network devices is often low and the complexity of the management
task is thus concentrated in the management infrastructure outside the managed system. This way of separation between the managed system and the management system has been justified with the arguments that it keeps away resource-intensive management tasks from running on network devices and that it establishes a clear separation of concerns between equipment manufacturers and providers of management solutions.

Over the past 20 years, the above paradigm has proved fairly successful when building networks of moderate size (say, below 1000 nodes), whose configurations rarely change, whose load patterns evolve slowly and which require outside intervention only within minutes or above. These assumptions, however, do not hold anymore for many of today’s networks and emerging networked systems, including modern access networks and data center environments, often with 10,000 or 100,000 managed elements within a single management domain. In addition to being large scale, these systems require fast reaction times in response to load changes and failures, sometimes within a subsecond, in support of mission-critical business processes.

One way to achieve scalable operation and rapid adaptation is to decentralize management tasks and to embed key management functions in the managed system itself. While such a system still maintains a central control point in each domain, the rate of interaction between the management system and the managed system can be significantly reduced when compared to a traditional management system. The objective of this chapter is to present new management protocols in support of such an approach.

In this chapter, we survey different approaches to distributed management and highlight an architecture that is especially suitable for embedded and scalable management. We introduce two classes of protocols for distributed management within such an architecture. These protocols execute as embedded functions inside the managed system and can perform a range of monitoring and control operations in support of the FCAPS management tasks. We restrict ourselves to protocols that use distributed trees. Other approaches, such as those using gossip protocols, are currently under investigation. They seem, however, less efficient in many cases and harder to make robust against failures when compared to tree-based protocols. Also, we limit the discussion to a single administrative domain.

2 Distributed Management

2.1 The Centralized Management Model

Fig. 1 shows the building blocks and their interactions of the basic centralized management model. On a conceptual level, management systems deployed today usually follow this model. We use SNMP terminology to explain it. The managed system includes a set of nodes, generally network devices, whose configuration and state data that is of interest to network management is maintained as managed objects by agents. The FCAPS management tasks are executed in programs that run on management stations or servers, which form the management system. (For simplicity, the figure shows a single management station, while in practice management systems often are large, complex systems themselves.) A management program can read and write managed objects through management protocols such as SNMP.
An interaction between the management system and the managed system is generally initiated by a management station, whereby a request from such a station sent to a node is followed by a response from the node. In the context of monitoring, this interaction pattern is called polling and referred to as pull-model. Alternatively, a node can initiate an interaction, e.g., to report an event to a management station, for which the term push model is used.

While the centralized management model given above has the advantage of design simplicity, it is not scalable, in the following sense. Assume program P on the management station polls the nodes for link state information and then recomputes the routes for network traffic. It is easy to see that the management traffic incurred by this operation grows linearly with the number of nodes, and the load on the management station, as well as the execution time of the operation, grows at least linearly with that number. There is therefore a limit to which the basic centralized model scales, which is around 1000 nodes in many settings.

2.2 Approaches to Distributed Management

One way to increase the scalability of centralized management is to distribute a management task, if this is possible, and to run it on a distributed management infrastructure. An example of such a solution can be found in traditional telecom networks, where trees of processing nodes are used for event processing and filtering.

Fig. 2 shows two alternative ways to increase scalability, both of which assume execution environments for processing management code on network nodes. The first is that of a script-enabled agent, whereby a management program P downloads a script S onto nodes of the managed system. S runs in an execution environment on a node and sends results back to P. This approach is also known as management by delegation [6] and is useful, e.g., to continuously gather and process statistical information from local state variables, which can trigger alerts that are sent to a management station. Typical languages to write such scripts in are Tcl/Tk and Java.
The right side of Fig. 2 shows a second approach called mobile agents. The main difference between both approaches is that, with script-enabled agents, a node communicates exclusively with the management station, while mobile agent programs can migrate from node to node, taking with them their execution states. (For an overview of mobile-agent concepts, see e.g. [16].)

2.3 An Architecture for Peer-to-Peer Management

Fig. 3 shows the architecture that supports the protocols for distributed management presented in the remainder of the chapter. It centers around the idea of a management plane, which conceptualizes the management resources inside the managed system. Each network node has an associated execution environment in the management plane, henceforth called management node, symbolized by a cube in the figure, which represents processing, memory and storage capacity. A management node has knowledge about other nodes in its neighborhood and can communicate with them through exchanging messages. This message network for peer interaction can be modeled as a connected graph, with the management nodes as vertices and neighbor relationships as edges, and we refer to it as the network graph. Management protocols in this architecture can be modeled as distributed algorithms that execute on this graph. They read and process state information in the management nodes and produce output that is available in one or more of them.

The role of the outside management system, represented in Fig. 3 through a management station, stays the same as that of the management models discussed above—it serves as the control point from which FCAPS tasks are initiated and their progress monitored. However, compared to centralized management, the rate of interaction between the management system and the managed systems is greatly reduced, since many functions, such as estimating global states and computing configurations can be performed by the management plane. As the management plane is part of the man-
The description of the above architecture is high level and thus has many possible realizations. A management node, for instance, can be realized as a virtual machine running (a) on a CPU inside a router, (b) on a blade that connects to a router backplane, or (c) in an appliance that is situated close to a router. All of these realizations are commercially available today. However, in today’s technology, a management node usually interacts only with the management station, following the model of script-enabled management outlined above, and thus lacks peer interaction, which characterizes this architecture. Depending on the specific realization of the management plane, the communication between a network element and its associated management node can take many forms, from using primitives for inter-thread communication to local SNMP interfaces.

The network graph on the management plane can be realized as an overlay. Similar to the management functions provided in the plane, the network graph can be constructed and maintained by distributed algorithms, for instance gossip algorithms (e.g., [14, 17]). As scalability is a design goal for this architecture, any realization of the management plane must be self-organizing and adaptive; the plane must auto-configure when the system is set up and must adapt to configuration changes and failures.

2.4 Comparing the discussed approaches

Fig. 4 summarizes and compares the outlined management approaches. The centralized management model offers design simplicity but lacks scalability and the ability to
react fast in response to network events. Script-enabled agents allow to mitigate these drawbacks of centralized management for certain tasks, at the expense of introducing execution environments into the managed system. Mobile agents allow the managed system to autonomously handle complex management tasks, thus in principle enabling scalability and fast reaction to events, which is bought by a higher complexity of the mobile agent execution environment and programs when compared to the previous approaches. Finally, peer-to-peer management in the context of the management plane achieves scalability through the use of distributed algorithms executing on the network graph. These algorithms are generally of low complexity (when measured in lines of code). The following sections provide examples of such algorithms.

### 3 Protocols for distributed management

In this section, we present two protocols that execute in the management plane of the peer-to-peer management architecture shown in Fig. 3. In the presentation, we distinguish between the protocol and its underlying distributed algorithms. The distributed algorithms capture the essence of the functionality from the point of view of algorithmic theory, while a protocol can be seen as an implementation of these algorithms, extended for practical use in a specific application domain. One such extension is the aggregator object, which encapsulates the management semantics of the protocol, such as the global state the protocol estimates for the purpose of monitoring or the value of a control parameter for the purpose of configuration. (In the distributed-systems literature, the terms distributed algorithm and protocol are often used with the same meaning.)

We assume that the algorithms execute on a connected network graph with bidirectional links. Each node on this graph has a global identifier, and it can distinguish its neighbors on the graph. Neighboring nodes can exchange messages, which are read in the order they are received. We assume an asynchronous execution model with bounded delays. The reader may consult a textbook on distributed algorithms for more details on execution models (e.g., [15, 10]).
3.1 The Echo Protocol

The echo protocol is a very versatile management protocol for networked systems. It is simple, generally very scalable and executes correctly on any connected network graph. It can be used for monitoring purposes, such as distributed polling, global state estimation, resource discovery and distributed configuration. Its potential application areas expand beyond traditional network management and include sensor networks and distributed computing systems.

The execution of the echo protocol can be understood as the subsequent expansion and contraction of a wave on a given network graph (see Fig. 5). The execution starts and terminates on an initiating node of the graph, also called the root (node). The wave expands through explorer messages, which nodes send to their respective neighbors. During the expansion phase, local operations are triggered on the nodes after receiving an explorer. The results of these local operations are collected in echo messages when the wave contracts, so that the aggregated result of the global operation becomes available at the root node. During the expansion phase, the protocol constructs a spanning tree on the network graph for the purpose of collecting and aggregating the partial results during the contraction phase.
message types:
1: (EXP, from)  \(\triangleright\) explorer sent by node from
2: (ECHO, from)  \(\triangleright\) echo sent by node from

data structures:
3: \(N := \) set of neighbors of root or node \(v\);

root node:
4: forall \(n \in N\) send (EXP, root) to \(n\);
5: while \(N \neq \emptyset\) do
6:  receive (ECHO, \(n\));
7:  \(N := N - \{n\}\);
8: end while
9: ‘Echo completed’;

non-root node \(v\):
10: receive (EXP, \(n\));
11: \(parent := n; N := N - \{parent\}\);
12: forall \(n \in N\) send (EXP, \(v\)) to \(n\);
13: while \(N \neq \emptyset\) do
14:  receive (ECHO, \(n\));
15:  \(N := N - \{n\}\);
16: end while
17: send (ECHO, \(n\)) to \(parent\);

Figure 6: The echo algorithm by Segall.

3.1.1 The Echo algorithm by Segall

The distributed algorithm underlying the echo protocol has been introduced by Segall [12]. Our presentation here is an adaption from [15].

Fig. 6 shows the code of the echo algorithm, which executes on the nodes of a connected, bidirectional graph \(G\). Two types of messages, explorer messages of the form (EXP, from) and echo messages of the form (ECHO, from) are exchanged between nodes. The variable from identifies the node that sends the message. The variable \(N\) is initialized with the set of neighbors of a node on \(G\) (line 3). The code of the root node, where an execution is initiated and terminated, is given in lines 4-9. All nodes except the root node execute the code in lines 10-17. The root node starts the execution by sending an EXP message to all its neighbors \(n \in N\) (line 4). After that, it receives an ECHO message from each neighbor \(n\), in any order (lines 5-8). Once the last message has been received, the execution on the root node is terminated (line 9). A non-root node \(v\) waits to receive an EXP message from one of its neighbors \(n \in N\) (line 10). The parent variable is then set to the sender of this message, and the sender is removed from \(N\) (line 11). Then, the node sends an EXP message to all neighbors, except the one identified in parent (line 12), and it receives an ECHO message from each of those nodes, in any order (lines 13-16). Finally, it sends an ECHO message to its parent, and the execution on node \(v\) terminates (line 17).
The echo algorithm creates a spanning tree (which we also call aggregation tree) on the network graph, with the initiating node being the root node of the tree and the parent variable on each non-initiator node pointing to its parent on the spanning tree. During the execution of the algorithm, a node sends an EXP message to each of its neighbors and receives a message of type EXP or ECHO from each neighbor. As a consequence, during the execution of echo, each link of the network graph is traversed by two messages, one in each direction. These and other properties of the algorithm are proved in [15].

Note that the algorithm solely performs a distributed synchronization function. For instance, no numerical value is computed as part of its execution. Note also that the algorithm relies only on local information in the form of knowledge about its direct neighbors of the graph (i.e., the set $N$). Neither the complete list of nodes in the system nor the system size is locally known. This is of practical importance, as the algorithm performs correctly after a change of the network graph between two consecutive runs (such as after adding or removing nodes), as long as the graph stays connected and each node has the correct information about its direct neighbors.

### 3.1.2 The echo protocol

We now extend the above algorithm into a protocol that is useful for network management purposes. First, we allow each node of the network graph to be the root node of an echo algorithm. Second, we allow the protocol to execute a local management operation on each node during the expansion phase and to aggregate the results of these local operations during the contraction phase, making the aggregate of all results available at the root node at the end of the execution. We achieve this by introducing a local aggregator object, which contains the state of the (distributed) management operation, together with methods that specify the local operation and the aggregation process.

Fig. 7 shows the code of the echo protocol. It executes on a connected, bidirectional network graph $G$. The aggregator object encapsulates the state and functionality of the distributed management operation. We refer to the state of the aggregator also as aggregate. The aggregator has four methods (lines 1-4). The method initiate() performs the local management operation and updates the state of the object. The method aggregate() aggregates the state with the value received through an echo message from a child node (see below). The method global() defines a function that is applied on the state. Finally, the method value() returns the state, i.e., the aggregate. Figs. 8, 9 include specific examples of aggregators and will be explained later. The echo protocol defines four types of messages that are exchanged between nodes during execution (lines 5-8). The message (INVOKE, invoker) is sent from a management station or a node of the management plane with id invoker (see Fig. 3). The recipient of the message initiates the echo operation on the management plane and becomes the root node of the aggregation tree. The message (RETURN, result) is sent from the root node to invoker with the result of the echo operation. The messages (EXP, from) and (ECHO, from, agg) serve the same purpose as in the echo algorithm (Fig. 6). The explorer message has an additional argument agg, which contains the aggregate of the subtree rooted at the sender node from. The procedure of the echo protocol combines
aggregator object $A$:

1: $A$.initiate() \(\triangleright\) initialize aggregate; perform local operation
2: $A$.aggregate() \(\triangleright\) aggregate the result from a child
3: $A$.global() \(\triangleright\) perform an operation on the aggregate (root node)
4: $A$.value() \(\triangleright\) return the current value of the (partial) aggregate

message types:

5: (invoke, invoker) \(\triangleright\) echo invoked by invoker
6: (return, result) \(\triangleright\) return result of echo operation
7: (exp, from) \(\triangleright\) explorer sent by sender
8: (echo, from, agg) \(\triangleright\) echo with result agg sent by sender

9: procedure ECHO( )
10: \(N:=\) set of neighbors of node $v$;
11: \(visited:=\)false;
12: while true do
13: \hspace{1em} receive message;
14: \hspace{2em} switch (message)
15: \hspace{3em} case (invoke, invoker):
16: \hspace{4em} $v$ is root
17: \hspace{4em} $A$.initiate();
18: \hspace{4em} if $N \neq \emptyset$ then \(\triangleright\) $v$ is only node in $G$
19: \hspace{5em} send (exp, $v$) to nodes in $N$;
20: \hspace{4em} else
21: \hspace{5em} $A$.global(); send (return, $A$.value()) to invoker;
22: \hspace{3em} end if
23: \hspace{2em} case (exp, from):
24: \hspace{3em} $N := N - \{from\}$;
25: \hspace{3em} if not visited then
26: \hspace{4em} visited := true; parent := from;
27: \hspace{4em} $A$.initiate();
28: \hspace{4em} if $N \neq \emptyset$ then
29: \hspace{5em} send (exp, $v$) to all nodes in $N$;
30: \hspace{4em} else
31: \hspace{5em} send (echo, $A$.value()) to parent; \(\triangleright\) $v$ is a leaf
32: \hspace{4em} end if
33: \hspace{3em} else
34: \hspace{4em} do nothing; \(\triangleright\) from is not neighbor of $v$ on tree
35: \hspace{3em} end if
36: \hspace{2em} case (echo, from, agg):
37: \hspace{3em} $A$.aggregate(agg);
38: \hspace{3em} $N := N - \{from\}$;
39: \hspace{3em} if $N = \emptyset$ then
40: \hspace{4em} if $v \neq$ root then
41: \hspace{5em} send (echo, $A$.value()) to parent;
42: \hspace{4em} else
43: \hspace{5em} $A$.global(); send (return, $A$.value()) to invoker;
44: \hspace{4em} end if
45: \hspace{3em} end if
46: \hspace{2em} end switch
47: \hspace{1em} end while
48: \hspace{1em} end procedure

Figure 7: The echo protocol. Pseudocode for node $v$. 

the code for the root node and non-root nodes (lines 9-47). It is executed on every node $v$ of the network graph $G$. The variable $N$ is initialized with the set of neighbors of node $v$ (line 10). A boolean variable $visited$, indicating whether the node has received an EXP message, is set to false (line 11). The while loop (lines 12-46) defines how the messages INVOKE, EXP and ECHO are processed. (The message RETURN is processed by the invoking node, which occurs outside the scope of the echo protocol.) Upon receiving the message ((INVOLVE, invoker), $v$ initiates the echo operation as root (lines 15-21). It performs initiate() and sends an EXP message to all its neighbors $n \in N$. In the (unusual) case where $G$ contains only a single node, the echo operation terminates and the result is sent to the invoking node invoker. Lines 22-34 define how an EXP message is processed by a non-root node (the root does not receive any EXP message). In case this is the first EXP message the node receives, it sets $visited$ to true. Also, the $parent$ variable is set to the sender of the message, and the sender is removed from $N$. Then, the node sends an EXP message to all neighbors, except the one identified in $parent$. In the case where $N = \emptyset$, which means that $v$ is a leaf of the aggregation tree, the node sends an ECHO message to its parent, and the execution on node $v$ terminates. Line 33 describes the situation where the EXP message originates from a neighbor on $G$ that is not a neighbor on the aggregation tree. In this case, no further action is taken. Lines 35-44 define how an ECHO message is processed. First, the local aggregate is updated with the aggregate $agg$ of the child $from$. Then, the node removes $from$ from $N$. Once $v$ has received an ECHO message from each of its children, i.e., $N = \emptyset$, it sends the aggregate to its parent and the execution on node $v$ terminates. In case $v$ is the root node, it performs global() and sends the result of the echo operation to the invoking node.

3.1.3 Echo-based management operations

We describe three types of operations that can be realized with the echo protocol.

1. Computing global functions of local variables. An important application of echo is distributed polling of network-wide aggregates. Examples of such aggregates include sums, averages or extremal values of local variables across all nodes of a networked system. The echo protocol performs this operation through tree-based, incremental aggregation of values in the management plane.

Formally, echo can compute a global function $F = F(x_1, \ldots, x_N)$ on local variables $x_i, i = 1, \ldots, N$, whereby each variable $x_i$ is associated with a node $v_i$ of a network graph $G = (V, E)$, with nodes $V$, $|V| = N$, and edges/links $E$. In the following, we give three characterizations of functions $F$ that can be computed in a single execution of the echo protocol.

- If $F$ can be written as a binary function $f$ that is both commutative (i.e., $f(x, y) = f(y, x)$) and associative (i.e., $f(x, f(y, z)) = f(f(x, y), z)$) for all $x, y, z$, then $F$ can be computed in a single execution of echo. An obvious example of such a function is sum. $f$ must be commutative, as we want the aggregate to be independent of the particular position of a node on the spanning tree. It must also be associative, as we want the aggregate to be independent of the order in which a node processes
incoming ECHO messages. Furthermore, if $F$ is of the form $F = g(f_1(), ..., f_k())$ whereby $f_i, i = 1, ..., k$, is a commutative and associate function on $(x_1, ..., x_N)$, and $g$ is a computable local function, then $F$ can be computed in a single execution of echo. The values $f_1(), ..., f_k()$ are computed through incremental aggregation as specified in aggregate($()$), while $g$ is computed on the root node and defined in global($()$). A simple example of such an $F$ is the average function, which not associative, as $\text{average} (\text{average}(1, 2), 3) \neq \text{average}(1, \text{average}(2, 3))$. Therefore, aggregation of average is performed using a vector $(f_1(), f_2())$, with $f_1 = f_2 = \text{sum}$ and $g = f_1/f_2$. (See example in Fig. 9).

- A global function $F$ on local variables $x_i$ is computable in a single execution of echo, if $F$ can be written as $F = g(N(m_1), ..., N(m_k))$, whereby $m_1, ..., m_k$ are the possible values for $x_i$, $N(m_i)$ is the number of occurrences of $m_i$ and $g$ is a computable local function. The values $N(m_1), ..., N(m_k)$ are computed through incremental aggregation as specified in aggregate($()$), while $g$ is computed on the root node and defined in global($()$). Examples of such functions $F$ include histograms of local variables, e.g., a vector showing the aggregate use of network resources by various applications, such as streaming video, peer-to-peer, etc.

- If we define aggregate($()$) to simply concatenate the local values $x_i$, then echo will provide the root node with a list of all local values and thus $F = F(x_1, ..., x_N)$ can be any computable function. Such a solution, however, is generally not feasible in a large networked system, as the size of an ECHO message increases linearly with the size of the subtree rooted at the sender node, and the load on the root node becomes the same as that on the management station in the centralized model (Fig. 1).

Note that in our formalization, we assume that the values of the local variables $x_i$ do not depend on time, i.e., they do not change during an execution of echo. From a management perspective, however, tracking global aggregates (i.e., global states) that change over time is important. To see why the protocol as presented above is still useful in practice, we note that the global control loop of a management system generally executes on a time scale of minutes or above, in exceptional cases on a time-scale of seconds. Consequently, we can assume (and observe in practice) that the values of $x_i$ change (or are sampled) at a rate of seconds or above. When comparing this figure with the expected execution time of echo, which is well below one second in large networks (e.g., [8]), it becomes clear that we can assume constant values for $x_i$ (or, equivalently, neglecting the execution time of echo) in many cases. For scenarios where this assumption does not hold, the protocol can be modified, for instance, in a manner that reading the local variables $x_i$ is delayed on all nodes until the expansion phase has globally concluded. (We assume here a sufficiently accurate synchronization of local clocks.)

We give two simple examples of echo-based management operations by presenting their respective aggregator objects. The first one, MaxLoad($()$), identifies in a network the link with the highest load. The second example, AverageLoad($()$), computes the average load over all network links. We assume here that each network device is associated with a node of the network graph, which can access its configuration and state variables. Fig. 8 shows the code of MaxLoad($()$). The state of the aggregator is the
vector \((\text{maxLoad}, \text{lmax})\), whereby \text{maxLoad} contains the maximum link load known to the node, and \text{lmax} identifies a link that carries that load (lines 1-2). The method \text{initiate()} initializes the state vector with data from the associated network device (lines 4-9). The method \text{aggregate()} aggregates the state vector with the aggregate from a child node (lines 10-15). The method \text{value()} returns the state vector (lines 16-18), while the method \text{global()} is empty, since \text{value()} already produces the result of the echo operation. The code of the aggregator \text{AverageLoad()}, given in Fig. 9, has the same structure as that of \text{MaxLoad()}. Note that in this case the method \text{global()} is not empty. It computes the global average over all link loads in the network.

2. **Network Search**  
As echo performs a complete (and parallel) traversal of the network graph during an execution, every node is visited and can be searched, say, for a local resource. This way, it is possible, for example, to identify the set of routers that run IOS version x.y and to make this set available at the root node. The local search function is defined in \text{initiate()}, and \text{aggregate()} specifies how the node ids are encoded before sending them up the aggregation tree. In contrast to using echo for computing global functions, aggregation of local results for network search generally includes concatenating ids of nodes that contain the resource of interest. As a consequence, the size of a message tends to grow from a leaf node towards the root node.

3. **Performing local operations on nodes with selected properties**  
Performing local control operations on nodes across a network relies, in the same way as network search, on the property of the echo protocol to completely traverse a network during an execution. An example of such an operation is ‘update module z on all routers that run IOS version x.y.’ In the aggregator object, the test of a node’s properties and the update operation are specified in \text{initiate()}, while the collection of the results is defined in \text{aggregate()}.

3.1.4 Performance of echo-based operation

Most performance metrics computed for the echo algorithm translate in a straightforward way to the extended echo protocol, if we assume upper bounds for communication delays between nodes and processing delays for local message processing.

- **Management traffic:** The execution of the protocol generates a balanced load on the network graph \(G\), with two messages traversing each link in opposite direction, which amounts to a total of \(2|E|\) messages. The size of these messages depends on the specific aggregation function. Message sizes can become large on links close to the root, if the aggregation function includes concatenating local results for instance. The number of messages generated is also referred to as message complexity in the context of distributed algorithms.

- **Processing load:** If we assume the load on a node to grow proportionally with the number of incoming messages that need to be processed, then the load increases proportionally with the number of neighbors the node has, i.e., with the degree of the node on the graph. We can say that the load per node grows \(O(\text{deg}(G))\),
1: **aggregator object** MaxLoad( )

2: var: maxLoad: int;  
   ▷ the maximum link load locally known

3: lmax: linkId;  
   ▷ link with maximum load locally known

4: procedure initiate( )
   var: L: linkIdSet;
   L := set of outgoing links;
   maxLoad := max\_link∈L load(link);
   lmax := link in L with value maxLoad;
end procedure

5: procedure aggregate([lchild: linkId; childLoad: int])
   if childLoad > maxLoad then
      lmax := lchild;
      maxLoad := childLoad;
   end if
end procedure

6: procedure value( )
   return ([maxLoad, lmax])
end procedure

7: procedure global( )
   empty;
end procedure

8: function load(l: linkId)
   return current load on link l;
end function

9: end object

**Figure 8:** Pseudocode for aggregator object MaxLoad()
1: **object** `AverageLoad()`
2: \[ \text{var: } sumLoad := 0; \quad \triangleright \text{total load of the (sub)tree rooted at local node} \]
3: \[ nLinks := 1; \quad \triangleright \text{number of network links of the (sub)tree} \]
4: \[ \text{procedure initiate( )} \]
5: \[ \text{var: } L : \text{linkIdSet}; \]
6: \[ L := \text{set of outgoing links}; \]
7: \[ sumLoad := \sum_{\text{link} \in L} \text{load(link)}; \]
8: \[ nLinks := |L|; \]
9: \[ \text{end procedure} \]
10: **procedure** aggregate([sumLoadChild: int; nLinksChild: int])
11: \[ \text{sumLoad} := \text{sumLoad} + \text{sumLoadChild}; \]
12: \[ nLinks := nLinks + nLinksChild; \]
13: \[ \text{end procedure} \]
14: **procedure** value( )
15: \[ \text{return} ([\text{sumLoad}, nLinks]) \]
16: \[ \text{end procedure} \]
17: **procedure** global( )
18: \[ \text{averageLoad} := \text{sumLoad}/nLinks; \]
19: \[ \text{end procedure} \]
20: **function** load(l: linkId)
21: \[ \text{return} \text{current load on link } l; \]
22: \[ \text{end function} \]
23: **end object**

Figure 9: Pseudocode for aggregator object `AverageLoad()`
whereby $\text{deg}(G)$ stands for the maximum degree of any node on $G$. The processing load is also referred to as computational complexity.

- *Execution time:* The execution time of an echo operation increases linearly with the height of the spanning tree, which is bounded by the diameter of the network graph, $\text{diam}(G)$. It also increases linearly with $\text{deg}(G)$. The execution time is also called time complexity in the context of distributed algorithms.

The topology of the network graph $G$ obviously influences the performance metrics of an echo-based operation. Graphs with small diameters and high degrees generally shorten the execution time by reducing the time used for communication, but they increase the processing load on some nodes. To take an extreme case where the graph has the topology of a chain, the execution time is $O(N)$, $N$ being the number of nodes, while the processing load is two messages per node (except for the two end nodes), independent of the choice of the initiating node. Considering another extreme where the graph is a star, the execution time is also $O(N)$, and the center node processes $N$ messages while all other nodes process just one.

For many applications, a network graph $G$ is preferred that has a fixed degree $\text{deg}(G)$ for all nodes and a diameter that increases with the logarithm of the system size. On such a graph, the echo protocol exhibits an execution time of $O(\log(N))$, a balanced processing load of $2 \times \text{deg}(G)$ messages per node and a balanced traffic load of two messages per link, independent of the choice of the initiating node. Note that such a performance profile contrasts with that of a management operation executed in the centralized model (Fig. 1), e.g., in an SNMP-based management framework where a management station communicates with all network devices through polling. In such a case, the processing load on the management station is at least is $O(N)$, as is the execution time of the management operation, while the traffic load experienced on the link that connects the management station with the network is $O(N)$ messages. This simple analysis supports the following experience: while in small networks and for specific network configurations a centralized management operation can be more efficient than an equivalent echo-based operation, in large-scale networks an echo-based operation can significantly outperform a centralized one. (A specific example can be found in [9].) The gain in scalability comes at the cost of a more complex management infrastructure, which entails a management plane with associated communication and processing resources within the managed system (see Fig. 3).

### 3.1.5 Extensions for practical applications

The echo protocol presented above is appropriate for gaining a basic understanding of its concepts and for performing a formal analysis, but it must be further extended and adapted for practical use. Here are some examples:

- *Concurrent execution* An invocation identifier can be introduced, which allows for running several echo operations simultaneously in the management plane, possibly with different initiating nodes.

- *Restricted scope.* The echo protocol executes on all nodes of a network graph, which spans the entire network (as we consider only a single domain). Such a
large scope of operation is often unwanted or simply not needed. We can restrict
the scope by introducing a ‘hop counter’ variable that bounds the expansion of
the protocol to a configurable number of hops from the initiating node. When
following this approach, we must consider that nodes with maximum hop count
from the root may be involved more than once in the same execution.

- **Stationary tree.** In a case where echo-based periodic polling from the same root
node is performed, it may be more efficient to apply a version of echo that keeps
the state of the the spanning tree alive between runs. Such a solution though
must maintain the tree in case the network graph changes.

- **Robust echo.** The presented version of the echo protocol is not robust to cer-
tain changes to the network graph that result from node churn or failures. As
mentioned above, if the network graph changes between two executions of echo,
the protocol executes correctly. If, however, a node fails (i.e., disappears from
the graph) during the contraction phase, it can happen that its parent waits to
receive an ECHO message from the failed node—an event that never occurs—and
the protocol thus deadlocks. A possible approach to deal with such crash failures
is to introduce an event, triggered either by a timeout or a failure detector, that
lets a waiting node resume protocol operation.

### 3.2 The tree-based GAP Protocol

#### 3.2.1 Design goals and design principles

The GAP protocol (GAP stands for Generic Aggregation Protocol) provides a man-
agement station (or a management node in the management plane) with a continuous
estimate of an aggregate that is computed over local variables across all nodes of a
networked system. The protocol dynamically adapts to node churn and node failures
in the sense that it continues to give accurate estimates after a brief transition pe-
riod following such events. GAP allows to control the tradeoff between the protocol
overhead and the accuracy with which the aggregate is estimated.

Formally, we consider a dynamically changing network graph \( G(t) = (V(t), E(t)) \),
in which nodes \( v_i \in V(t) \) and edges/links \( e_j \in E(t) \subseteq V(t) \times V(t) \) appear and dis-
appear over time. Each node \( v_i \in V(t) \) has an associated local variable \( x_i(t) \). The
GAP protocol executes on \( G(t) \) and continuously computes a global function
\( F(t) = F(x_1(t), ..., x_N(t)) \) on the local variables \( x_i(t), i = 1, ..., N(t), \) with \( |V(t)| = N(t) \). The
result of the computation is available on a distinguished root node of \( G \).

While GAP is a monitoring protocol only, the echo protocol can be applied to other
tasks as well, as discussed above. Restricted to the context of monitoring, the main
difference in functionality between both protocols is this. A single execution of echo
provides an estimate of an aggregate at a specific point in time. Repeated executions
give a sequence of snapshots of the aggregate over time. Consequently, the value of the
aggregate between two snapshots is not known. In contrast to echo, an execution of
GAP, once started, continues until it is terminated. GAP thus provides a continuous
estimate of the aggregate.

Similar to the echo protocol, GAP creates a spanning tree on the network graph,
which is used to perform incremental, distributed aggregation, with the result becoming available at the root node. In contrast to echo, which follows the pull-model for reading local variables, GAP is a push protocol in the sense that updates to the local variables are ‘pushed’ upwards the tree, from the leafs towards the root. Since GAP executes continuously, it needs mechanisms that maintain the spanning tree, in order to cope with node churn and failures.

GAP allows controlling the protocol overhead by limiting the message rate on the network graph and thus along the links of the spanning tree. This is an effective way to bound both the processing overhead on nodes of the network graph and the communication overhead on its links. The price for a reduced overhead is generally an increased error in estimating the aggregate.

The description of GAP presented here is adapted from [4].

3.2.2 Underlying algorithms

Three distributed algorithms underlie the GAP protocol. The first is a distributed version of the well-known Belman-Ford algorithm, which constructs a BFS (Breath-First Search) spanning tree on a connected network graph with a distinguished root node [10]. A BFS tree has the property that it connects each node to the root with a shortest path, whereby the distance is measured as the number of edges of the path.

Fig. 10 shows the algorithm, which executes on the nodes of a connected, bidirectional graph $G$. Each node maintains a level variable that indicates its distance to the root and a pointer to its parent node. The algorithm builds a spanning tree in a distributed fashion, starting from the root and continuing towards the leafs. The tree is encoded in the parent variables. (level and parent actually contain the belief of the node, not necessarily the correct values; see comment below.) During the execution of the algorithm, nodes exchange messages of the form (UPDATE, $n$, level), which convey that node $n$ has distance level from the root. The code of the root node is given in lines 2-3. The root node starts the execution by setting its level and parent variables to 0 and root, respectively. Then it sends an UPDATE message to all its neighbors, indicating its level. A non-root node $v$ executes the code in lines 4-11. It initializes its level and parent variables with infinite and undefined, respectively. Then, it performs an infinite loop, reading and processing an UPDATE message from any neighbor $n$ during each iteration. If the message indicates that $v$’s level is larger than that of its neighbor plus 1, then $v$ updates its level and sets $n$ as its new parent.

The algorithm guarantees that the variables level and parent eventually contain correct values, i.e., once no more messages are exchanged. Before this time, one can say that these variables contain the belief of a node regarding its distance to the root and its parent on the tree. The algorithm has a time complexity of $O(diam(G))$ and a message complexity of $O(|V||E|)$ in an asynchronous model [10]. Based on this algorithm, GAP builds up the spanning tree during initialization.

The second algorithm, developed by Dolev, Israeli and Moran [5], can be understood as an extension of the distributed Belman-Ford algorithm. The algorithm in [5] is self-stabilizing in the following sense. Assuming the root has the correct level, then, independent of a node’s initial values of the parent and level variables, the system converges to a state in which the parent pointers form a BFS tree, and the level vari-
**messages:**
1. (UPDATE, n, l);  \(\triangleright\) node n has distance l from root

**root node:**
2. level := 0; parent:=root;
3. send (UPDATE, root, 0) to all neighbors on G;

**non-root node v:**
4. level :=infinite; parent :=undef;
5. while true do
6. read (UPDATE, n,l);
7. if (level > l + 1) then
8. level := l + 1; parent := n;
9. send (UPDATE, v, level) to all neighbors on G except parent;
10. end if
11. end while

Figure 10: A distributed version of the Belman-Ford algorithm, which creates a BFS tree on a connected graph G with a distinguished root node.

**messages:**
1. (UPDATE, n, l);  \(\triangleright\) node n has distance l from root

**root node:**
2. level := 0; parent:=root;
3. while true do
4. send (UPDATE, root, 0) to all neighbors on G;
5. end while

**non-root node v:**
6. while true do
7. read (UPDATE, n,l);
8. level := l;
9. Among all neighbors of v with the smallest level, choose node with the smallest index k
10. level := level_k + 1; parent := k;
11. send (UPDATE, v,level) to all neighbors on G;
12. end while

Figure 11: A self-stabilizing algorithm by Dolev, Israeli and Moran, which creates a BFS tree on a connected graph G with a distinguished root node.
ables contain the correct distance to the root (cf. [5]). Fig. 11 shows the pseudocode. We describe specifically the difference to the code in Fig. 10. Instead of the root node sending a single \textit{update} message with its level to all its neighbors (Fig. 10, line 3), the root keeps sending such messages in an infinite loop (line 3-5). The reason for this is that, for non-root nodes, the variables \textit{level} and \textit{parent} are not initialized, as in Fig. 10, line 4. In fact, \textit{level} can start with any integer $> 0$ and \textit{parent} with any neighbor. Further, contrary to Fig. 10, each non-root node, maintains a variable \textit{level}_{n} for each neighbor $n$ on $G$. The neighbors of a node are identified through local indices, and, among the neighbors with minimal level, the one with the smallest index is chosen as parent (line 9). Finally, in the first algorithm, a non-root node sends an \textit{update} message to all neighbors except \textit{parent} (Fig. 10, line 9), while in this algorithm the \textit{update} message is sent to \textit{parent} as well (Fig. 11, line 11), the reason being that in this algorithm a node has knowledge about the level of its neighbors.

In [5] the algorithm in Fig. 11 is given for a shared memory model; we present it here for a message-passing model. GAP uses the idea behind the self-stabilizing property of this algorithm to maintain the BFS tree in response to node churn and node failures.

The third algorithm that underlies GAP is straightforward and enables the protocol to perform incremental, in-network aggregation along the spanning tree created and maintained by the above algorithms. The basic idea is that each node holds the aggregate of the subtree rooted at that node. Upon a change of its aggregate, the node sends a message with the new value to its parent, which processes the message and updates its own aggregate. Fig. 12 shows a sample spanning tree, as well as local values and aggregates of nodes, whereby \textit{sum} is used as the aggregation function.

### 3.2.3 The GAP Protocol

Nodes executing the GAP protocol keep local state related to the topology of the spanning tree (which we also refer to as aggregation tree) and the aggregation process. For this purpose, each node maintains a neighborhood table $T$ with an entry for itself and one for each neighbor on the network graph $G$. A table entry has the form $(\text{nodeId}, \text{status}, \text{level}, \text{aggregate})$. The \textit{status} field gives the relative position of a node on the aggregation tree, with values self, parent, child and peer. (peer refers to a neighbor on $G$ that is not a neighbor on the spanning tree.) The \textit{level} field indicates (a node’s belief of) the distance (in hops) from the node to the root, while the \textit{aggregate} field contains the aggregate of the subtree rooted at the node. Fig. 13 gives an example of a neighborhood table.

Fig. 14 shows the methods defined on the neighborhood table. The method \text{addEntry}(n, s, l, a) inserts a row for node $n$ with fields \textit{status} := $s$, \textit{level} := $l$ and \textit{aggregate} := $a$. The method \text{removeEntry}(n) removes the entry for node $n$ from the table. The method \text{updateEntry}(n, l, p, a) updates the entry for node $n$ according to the given values $l$, $p$, and $a$. Regarding the value for \textit{status}: if node $n$ is parent to the local node $v$, then the \textit{status} field remains set to parent, if $p = v$, then \textit{status} is set to child, otherwise \textit{status} is set to peer. The method \text{updateVector}() returns the vector $(\text{level}, \text{parent}, \text{aggregate})$ with values for the local node $v$. Finally, the method \text{restoreTableInvariant}() ensures that the node $v$ has a parent with minimum level among
Figure 12: Incremental aggregation on a spanning tree. The aggregation function is sum in this example.

<table>
<thead>
<tr>
<th>nodeid</th>
<th>status</th>
<th>level</th>
<th>aggregate</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>self</td>
<td>2</td>
<td>40</td>
</tr>
<tr>
<td>4</td>
<td>parent</td>
<td>1</td>
<td>70</td>
</tr>
<tr>
<td>7</td>
<td>peer</td>
<td>2</td>
<td>10</td>
</tr>
<tr>
<td>14</td>
<td>child</td>
<td>3</td>
<td>10</td>
</tr>
<tr>
<td>15</td>
<td>child</td>
<td>3</td>
<td>10</td>
</tr>
<tr>
<td>16</td>
<td>child</td>
<td>3</td>
<td>10</td>
</tr>
</tbody>
</table>

The numbers on the overlay show the node ids. The neighborhood table for node 6 is shown. The update vector of the table is (2, 4, 40). The aggregation function is sum. Each node has a local variable with value 10.

Figure 13: The neighborhood table (left) for node with id 6 of the network graph with overlaid spanning tree (right).
GAP neighborhood table:

\( T \): table with rows \((node, status, level, aggregate)\)

**table methods:**

- `addEntry(n, s, l, a)` \(\triangleright\) add entry for node \(n\)
- `removeEntry(n)` \(\triangleright\) remove entry for node \(n\)
- `updateEntry(n, l, p, a)` \(\triangleright\) update entry for node \(n\)
- `updateVector() : (l, p, a)` \(\triangleright\) give \(level, parent, aggregate\) of local node
- `restoreTableInvariant()` \(\triangleright\) maintain BFS property of spanning tree

Figure 14: GAP neighborhood table with its methods.

1: Among all neighbors of \(v\) with the smallest level, choose node with the smallest index \(k\)
2: \(level_v := level_k + 1; parent := k;\)

Figure 15: A possible implementation of `restoreTableInvariant()` on node \(v\).

its neighbors and that its own level equals that of its parent plus 1. A possible implementation, following the code in Fig. 11, is given in Fig. 15. (Since GAP initializes the `level` and `parent` variables with the value `undef`, we set `min(k, undef) = k` and `min(undef, undef) = undef`. The second line in Fig. 15 is executed, if `level_k \neq undef`.) The purpose of this method is to maintain the structural integrity of the spanning tree and its BFS property.

Note that the values of `status`, `level` and `aggregate` represent local states in a dynamic environment with delayed information. In this sense, they refer to a node’s current belief of its own level and that of its neighbors, for instance. Once no further changes to the system occur, the execution of the GAP protocol makes these values converge to globally correct values.

Using the self-stabilization property of the underlying algorithm (Fig. 11) one can show that the `status`, `level`, and `aggregate` fields of the neighborhood tables can be changed to any legal value, upon which the protocol converges to a BFS tree in finite time, with the correct (global) aggregate available at the root node, as long as all nodes keep an entry with `status = self` in their neighborhood table and the root node has the correct entry for itself.

The aggregator object \(A\) (Fig. 16) encapsulates the functionality of the distributed monitoring operation and allows to incrementally compute a global aggregation function on the aggregation tree. The object has three methods. The method `initiate()` starts a local service that produces a `LOCALVAR` message whenever the local variable changes its value (see below.) The method `aggregate()` computes the aggregate of the subtree rooted at the node, using the data in the neighborhood table, together with the local variable. The method `global()` defines a function that is applied on the node’s aggregate. In the example where the aggregation function is sum, \(A.aggregate()\) simply sums up the partial aggregates from the children and the local variable, and \(A.global()\) is empty.

Fig. 17 shows the code of the GAP protocol, which executes on the nodes
Aggregator object $A$

- $A$.initiate(): initiate update messages of change to local variable
- $A$.aggregate(): compute the (partial) aggregate of this node
- $A$.global(): perform an operation on the aggregate (root node)

Figure 16: The aggregator object for the GAP protocol.

of a connected, bidirectional graph $G$. Four types of messages are defined for this protocol (lines 1-4). The message (new, $n$) indicates the discovery of a new neighbor $n$, while the message (fail, $n$) reports that neighbor $n$ has failed. The message (update, $n$, $l$, $p$, $a$) communicates a state update by node $n$. Finally, the message (localvar, $x$) reports an update to the local variable $x$. When started on a node $v$, the protocol initializes the neighborhood table (lines 6-12), computes the vector (level, parent, aggregate) for node $v$ (line 13) and sends this vector to all known neighbors via an update message (line 14). Then, the node initiates the service that generates localvar messages, in order to receive updates to the local variable (line 15). Lines 16-37 contain an infinite loop, whereby a message is read and processed during each iteration. In case of a new message, an entry is created in the neighborhood table and an update message is sent to the newly discovered neighbor (lines 19-21). If a fail message is received, the entry for the corresponding node is removed from the neighborhood table (lines 22-23). A localvar message triggers an update of the node’s aggregate in line 30. Upon receiving an update message, the entry of the sender node is updated (lines 26-27). After the above statements have been executed, restoreTableInvariant() is invoked, the aggregate is updated, and the vector (level, parent, aggregate) for node $v$ is computed (lines 29-31). In case this vector has changed during the current iteration of the loop, the vector is sent to all neighbors via an update message (line 33).

The fact that an update is sent to neighbors only when the update vector changes (line 32) makes the GAP protocol more efficient than the algorithm in Fig. 11. One can further increase the efficiency of the version of GAP given here, without sacrificing correctness, if an update caused by a change in a node’s aggregate is sent only to its parent, instead of to all its neighbors.

To keep the code simple, Fig. 17 does not include any restriction on the rate at which a node sends update messages to its neighbors. For practical implementations of GAP, however, such a mechanism is needed for controlling the protocol overhead. For this reason, the version of the GAP protocol described in [4] includes a control parameter that limits to $r$ messages per sec the number of updates a node can send over a link of the network graph. It is not difficult to engineer the protocol in such a way that this parameter can be changed at run-time.
messages:
1: (NEW, n) \quad \triangleright \text{ new neighbor } n \text{ detected}
2: (FAIL, n) \quad \triangleright \text{ neighbor } n \text{ failed}
3: (UPDATE, n, l, p, a) \quad \triangleright \text{ node } n \text{ has aggregate } a, \text{ level } l, \text{ parent } p
4: (LOCALVAR, x) \quad \triangleright \text{ the local variable has value } x

5: procedure GAP( )
6: \quad T := \text{empty table};
7: \quad \text{if } v = \text{root} \text{ then}
8: \quad \quad \text{addEntry}(root, parent, -1, \text{undef});
9: \quad \quad \text{addEntry}(root, self, 0, \text{undef});
10: \quad \text{else}
11: \quad \quad \text{addEntry}(v, self, \text{undef}, \text{undef});
12: \quad \text{end if}
13: \quad \text{vector} := \text{updateVector}();
14: \quad \text{send (UPDATE, } v, \text{vector) to all neighbors;}
15: \quad \text{A.initiate();}
16: \quad \text{while true do}
17: \quad \quad \text{read message;}
18: \quad \quad \text{switch (message)}
19: \quad \quad \quad \text{case (NEW, from):}
20: \quad \quad \quad \quad \text{addEntry(from, peer, undef, undef);}
21: \quad \quad \quad \quad \text{send (UPDATE, } v, \text{vector) to from;}
22: \quad \quad \quad \text{case (FAIL, from):}
23: \quad \quad \quad \quad \text{removeEntry(from);}
24: \quad \quad \quad \text{case (LOCALVAR, x):}
25: \quad \quad \quad \quad \text{empty;}
26: \quad \quad \quad \quad \text{case (UPDATE, from, level, parent, aggregate):}
27: \quad \quad \quad \quad \quad \text{updateEntry(from, level, parent, aggregate);}
28: \quad \quad \quad \text{end switch}
29: \quad \quad \text{restoreTableInvariant();}
30: \quad \quad \text{A.aggregate(); if } (v = \text{root}) \text{ then A.global();}
31: \quad \quad \text{newvector} := \text{updateVector}();
32: \quad \quad \text{if } \text{newvector} \neq \text{vector} \text{ then}
33: \quad \quad \quad \text{send (UPDATE, } v, \text{newvector) to all neighbors;}
34: \quad \quad \quad \quad \text{vector} := \text{newvector;}
35: \quad \quad \text{end if}
36: \quad \text{end while}
37: \text{end procedure}

Figure 17: The GAP protocol, performing continuous aggregation of local variables on a network graph \( G \). The aggregate is available at the root node. Pseudocode for node \( v \).
3.2.4 Performance of the GAP protocol

Given upper delay bounds for sending a message between neighboring nodes and processing a message on a node, GAP’s performance metrics can be obtained from analyzing the underlying algorithms. This leads to the following statements.

- **Management traffic and processing load**: For the version of GAP with rate control, mentioned above, the management traffic (also called message complexity) is limited to a rate of \( r \) messages per sec for each link on the network graph \( G \). Furthermore, the maximum possible processing load on a node, measured in incoming messages per sec, increases proportionally with the degree of a node (i.e., the number of its neighbors). Consequently, the processing load on any node is \( O(r \times \text{deg}(G)) \). The rate \( r \) can therefore be used to control both the management traffic and processing load in the management plane. Also, the maximum possible traffic load per node and the maximum possible processing load per node are independent of the network size (for graphs with the same degree), which makes GAP suitable for large-scale networked systems.

- **Time for initialization, update of aggregate and reconfiguration due to node churn or failure**: The time it takes from starting the protocol on all nodes to the root having the correct aggregate is proportional to the height \( h \) of the aggregation tree and thus proportional to the diameter \( \text{diam}(G) \) of the network graph \( G \) (assuming that the local values do not change during initialization). To see this, note that \( h \) rounds are needed for the node with the longest distance from the root to have the correct level information, and it takes an additional \( h \) rounds for an update from that node to reach the root, in form of a chain of update messages from a leaf to the root. (A round here is the inverse of the message rate.) Furthermore, any update to a local variable will trigger an update of the aggregate on the root node within \( h \) rounds. Using similar reasoning, one can show that after adding a node to or removing a node from \( G \), it takes at most \( 2 \times \text{diam}(G) \) rounds, until the spanning tree has been adapted to the new topology.

- **Dependence on the network graph**: The performance of the GAP protocol depends on the topology of the network graph \( G \), in a similar way as the performance of the echo protocol does (see section 3.1.4). The traffic and processing loads, as well as the initialization, update and reconfiguration times, depend on the topology of the spanning tree, which, in turn depends on \( G \). For instance, if we choose a network graph with \( \text{diam}(G) = O(\log(|G|)) \), then the initialization time, the update time and the reconfiguration time of GAP are all \( O(\log(|G|) \times \text{deg}(G)) \), i.e., they increase with the logarithm of the network size, assuming \( \text{deg}(G) \) is bounded. (We use the word “choose” to emphasize that in many networked systems the network graph \( G \) can be constructed, e.g., by means of an overlay, and is not determined by the
3.2.5 Extensions for practical applications

While the pseudocode of GAP in Fig. 17 contains a complete protocol, it should be regarded as a skeleton for a practical implementation. Here are some issues that must be addressed when the protocol is implemented for an operational environment. For most of them simple solutions can be found that work effectively in practical scenarios—at the expense, of course, of making the protocol more complex.

- **Invocation parameters:** Support is needed to invoke the protocol with a list of parameters that, e.g., identify the root node of the aggregation process, the aggregator to be used, the local variable(s) to be aggregated, the network scope within which aggregation should be performed, the maximum message rate $r$ for communication between neighboring nodes, etc. In addition, the protocol as given in Fig. 17 must be extended to enable multiple concurrent invocations and to include a mechanism for terminating an invocation.

- **Robustness to node churn and crash failures:** Since GAP is self-stabilizing, the protocol is robust to node churn and crash failures, as long as the network graph stays connected when a node leaves or crashes. (An exception is the root node, which can neither leave nor fail. See below.) In such a case, the protocol as given in Fig. 17 repairs the spanning tree, if needed, and reinstates the BFS property. Three issues, however, need further attention. First, during the transition phase when the tree reconstructs, significant errors in estimating the aggregate can occur, especially if changes in the tree topology take place close to the root. Techniques can be devised to mitigate such estimation errors, for instance, by nodes signaling to the root when they detect churn or crash events. Second, an extension of GAP is needed to handle root failures. An obvious approach is to compute the global aggregate on several nodes at the same time, and to initiate a leader-election process when the designated root fails. Third, the case of partitioning of the network graph must be handled. As a consequence of nodes leaving or failing, the network graph can split up into two or more subgraphs that are not connected to one another. The protocol as presented in Fig. 17 runs correctly on the subgraph that contains the root. On other subgraphs, however, the levels of the nodes do not converge but keep increasing indefinitely.

- **Synchronized aggregation.** The GAP protocol accurately estimates aggregates assuming that computational and communication delays can be neglected when computing the global aggregate. The argument for these assumptions is that the sampling of local variables $x_i$ is typically performed in the order of seconds, while the other delays are often measured in milliseconds. If these assumptions do not hold, the computation of the global
aggregate can incur significant errors. To avoid such errors, GAP can be extended in such a way that (a) local values are sampled at global times $t_j$ at rate $r$ and (b) each node $i$ holds a vector with components $\text{aggregate}_{i,j}$, the value of the aggregate on node $i$ at time $j$. At time $t = t_j$ node $i$ keeps the vector $[\text{aggregate}_{i,j-w}, ..., \text{aggregate}_{i,j}]$, whereby $w$ is the height of the aggregation tree and can be chosen as $w = \text{diam}(G)$. An accurate estimate of the aggregate at the root node is then $\text{aggregate}_{\text{root},j-w}$, which means that the global aggregate is known with delay $w/r$.

- **Distance metrics other than hop count**: Based upon properties of the underlying algorithms (Figs. 10 and 11), GAP keeps executing correctly when the distance metric, which determines a node’s level value, is changed. If, for instance, link delay is chosen as metric instead of hop count, then the BFS property of the spanning tree translates into the fact that, for each node, the path on the tree to the root has minimal delay.

- **Global aggregate available on all nodes**: GAP computes the global aggregate at the root node only; all other nodes compute the (partial) aggregate of the subtree rooted at that particular node. It is possible to extend the GAP aggregation mechanism in a straightforward way—using the idea that each node on the spanning tree can be considered to be the root of an aggregation tree—, so that the global aggregate becomes available on all nodes of the network graph. Such a solution is more elegant and gives shorter update times than the naive approach whereby the root broadcasts to all nodes updates to the global aggregate.

## 4 Extensions of Echo and GAP Protocols

In the previous section, we presented two fundamental and complementary classes of protocols for distributed management—Echo and GAP. We gave a detailed account of the distributed algorithms that underlie those protocols, in order to let the reader better understand the code and the properties of the protocols. To keep the code modular, we introduced the aggregator object, which defines the local operation and the aggregation of the results, thereby encapsulating the semantics of the management operation. We gave examples of simple echo aggregators in figures 8 and 9, and we formally characterized echo aggregators for global state estimation in section 3.1.3. An instance of a more complex echo aggregator has been developed for the Weaver prototype system, which performs network-wide flow monitoring in real-time through an SQL-based interface [8, 9].

We mentioned in the previous section that the code given for the echo and GAP protocols should be regarded as a skeleton, which must be extended for a practical implementation. Such extensions primarily relate to software issues. More importantly though, both protocols can be used as building blocks for more
advanced or higher-level functions. In the following, we outline two examples of GAP extensions, both of which aim at reducing protocol overhead while achieving certain objectives. They inherit from GAP the functionality of creating and maintaining the aggregation tree and that of incremental aggregation.

The first such extension, named the A-GAP protocol, employs a local filter scheme, which prevents a node from sending an update to its parent when only a small change to its aggregate has occurred. Like the GAP protocol, A-GAP performs continuous monitoring of global aggregates, but, unlike GAP, it aims at minimizing the protocol overhead while achieving a configurable accuracy objective (such as a bound on the average absolute error). The local filters are computed in a distributed way, based on a stochastic model of the monitoring process. This model is computable in real-time as part of the protocol execution and allows predicting certain performance metrics, including overhead and estimation error. A-GAP thus allows, at runtime, to control the tradeoff between estimation accuracy and protocol overhead. A thorough presentation of the A-GAP protocol can be found in [11]. (Filter computation in A-GAP assumes the aggregation function to be sum. A related work, which investigates histogram aggregation instead of sum, is described in [7]).

A second extension of GAP, called the TCA-GAP protocol, detects threshold crossings of global aggregates in an efficient way. It applies the concepts of local thresholds and local hysteresis, aimed at reducing protocol overhead whenever the aggregate is “far” from a given threshold while ensuring correct detection. Similar to filter computation for A-GAP, the local thresholds for this protocol are computed in an asynchronous, distributed way. A detailed presentation of TCA-GAP can be found in [18]. A similar scheme for detecting threshold crossings has been developed by Breitgand et al. in the context of estimating the size of multicast groups [2].

5 Acknowledgements

Fetahi Wuhib and Misbah Uddin at KTH provided comments to an earlier draft, and helped with proof-reading and the preparation of the document.

6 Bibliography

[1] 4WARD Partners. The EU 7th framework project 4WARD. http://www.4ward-project.eu/.


