Developing Pattern-Based Management Programs

Koon-Seng Lim and Rolf Stadler

Center for Telecommunications Research and
Department of Electrical Engineering
Columbia University
New York, NY 10027-6699
{koonseng,stadler}@ctr.columbia.edu
http://www.comet.columbia.edu/adm

Abstract. We have recently proposed a novel approach to distributed management for large-scale, dynamic networks. Our approach, which we call pattern-based management, is based on the methodical use of distributed control schemes. From an engineering point of view, our approach has two advantages. First, it allows for estimating the performance characteristics of management operations in the design phase. (This was the subject of an earlier paper.) Second, it reduces the complexity of developing scalable, distributed management programs, by promoting the re-usability of key software components. In this paper, we demonstrate how pattern-based management programs can be designed and implemented. We propose an object model for these programs and give a concrete example of how a management task, such as obtaining the current link load distribution of the entire network, can be realized. The example illustrates that our model facilitates writing the key software components in a compact and elegant way.

1 Introduction

In our recent research, we have proposed a novel approach to distributed management, which applies particularly well to large-scale networks and to networks that exhibit frequent changes in topology and network state [5][6]. We call this approach pattern-based management. It is based on the methodical use of distributed control schemes. The key concept is that of *navigation patterns*, which describe the flow of control during the execution of a (distributed) management program. A navigation pattern determines the degree of parallelism and internal synchronization of a distributed management operation.

The concept of navigation patterns has two main benefits. First, it allows for the analysis of management operations with respect to performance and scalability [6]. Second, from a software engineering point of view, navigation patterns allows us to separate the semantics of a management operation from the control flow of the operation. This means that a typical management operation can be realized using different navigation patterns, which can be chosen according to different performance objectives. Also, since a pattern is generic and does not encapsulate any management-specific semantics, the same pattern can be used for different management tasks. We believe that, as a consequence, our approach will free the management application programmer from developing distributed algorithms, allowing him/her to focus on the specific management task and to select a navigation pattern that captures the requirements for that task.

Table 1: Management paradigms with different control schemes and programming models

Manager-Agent	Centralized control	Program runs on management station, operates on 'global' MIB
Management by Delegation, Remote Evaluation	Centralized control Dynamic delegation of subtasks	Program runs on management station, subprogram runs on local MIBs
Mobile Agent	Decentralized control, realized by intelligent autonomous decisions	Agent program controls code migration and operations on local MIBs
Pattern-based Management	Decentralized control, realized by network algorithms	Distributed program runs on topology graph whose nodes are local MIBs

Table 1 shows how pattern-based management relates to other management paradigms. The manager-agent model, based on the client-server paradigm, realizes a centralized control scheme. This control scheme can be characterized by a management program running on a management station and operating on the Management Information Base (MIB), which is distributed among agents in the network. The manager-agent model serves as a basis for the management standards behind SNMP and CMIP [15][9]. The main limitation of this model is its poor scalability, since it can lead to a large amount of management traffic, a large load on the management station, and long execution times for management operations, especially in large networks [17]. To overcome this drawback, the concepts of *management by delegation* (MbD) and *remote evaluation* (REV) were proposed [1][16]. These concepts allow for delegating tasks from the management station to the agents in the network via the downloading of scripts. A further step towards decentralized control was taken with the introduction of *mobile agents* for management tasks [11]. Mobile agents can be characterized by self-contained programs that move in the network and act on behalf of a user or another entity. Mobile agents are generally complex, since they exhibit some kind of intelligent behaviour, which allows them to make autonomous decisions. In contrast to the mobile agent approach, our paradigm, pattern-based management, stresses the use of simple, dedicated components, which run in parallel on a large number of network nodes. All of the above paradigms, except for the manager-agent model, necessitate an execution environment on the network nodes, which executes scripts, mobile agents or pattern programs, respectively.

Our work is close to Raz and Shavitt [12], which advocates the use of distributed algorithms in combination with active networking for network management. We differ from [12] in our emphasis on the separation of the flow of control of the management operation from its semantics through an explicit interface.

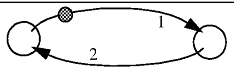
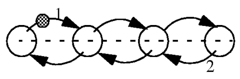
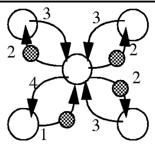
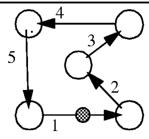
operation	typical application	navigation pattern
type 1: node-to-node	1 node control/monitor (e.g. get/set of variables)	
type 2: visit all nodes along a path/flow	1 flow/path control (e.g. traceroute, bottleneck detection, signalling, VPN operation)	
type 3:distribute agents to all nodes in subnet (parallel control)	subnet control, message broadcast (e.g. congestion location detection)	
type 3: visit all nodes in subnet (sequential control)	subnet control (e.g. topology detection)	

Fig. 1. Examples of simple navigation patterns

Figure 1 shows simple examples of navigation patterns. The most basic pattern is the type 1 pattern, where control moves from one node to another and returns after triggering an operation. This pattern exemplifies a typical manager-agent interaction. A type 2 pattern illustrates the case where control moves along a path in the network, triggers operations on the network nodes of this path, and returns to the originator node along the same path. A possible application of this pattern is resource reservation for a virtual path or an MPLS tunnel [13]. In a type 3 pattern, control migrates in parallel to neighbouring nodes, triggers operations on these nodes, and returns with result variables. This pattern can be understood as a parallel version of the type 1 pattern. Finally, in a type 4 pattern, control moves along a circular path in the network. All these examples illustrate that navigation patterns can be defined independently of the management semantics.

To fully exploit the advantages of pattern-based management, we envision the use of patterns with more inherent parallelism than those given in Figure 1. For instance, in [6] we have introduced the echo pattern, which triggers local operations on all nodes of the network and aggregates the local results in a parallel fashion.

As discussed in [6], navigation patterns can be defined using asynchronous parallel graph traversal algorithms, also known as network algorithms [8]. Note that the concept of a navigation pattern is very different from that of a design pattern as used in software engineering. While a navigation pattern captures the flow of control of executing a distributed operation, a design pattern describes communicating objects and classes for the purpose of solving a general design problem in a particular context [4].

A pattern-based management system can be realized in various ways. One possibility is through a mobile agent platform that is installed on the network nodes (or on control processors that are attached to the network nodes) and provides access to the nodes' control and state parameters through local interfaces [5]. On such a platform, management operations are coded as mobile agent programs, and navigation patterns define the migration, cloning and synchronization of the agents, which are executed on network nodes, perform local operations, and aggregate results. Another possibility for realizing the concept of navigation patterns is to program patterns on an active management platform, such as ABLE [12]. Such a platform includes commercial routers that have been enhanced by an active management node, which processes active packets that have been filtered out by the router. In ABLE, such packets contain Java code that is executed on the management node. An interesting, third possibility is the case whereby the code for performing a management operation is not transported to the nodes as part of this operation. (The code has either been installed beforehand or is loaded on demand from a code server.) In this case, pattern-based management can be realized by a system that exchanges messages between network nodes, which simply contain part of the execution state of the management operation.

This paper reports on our research in how to program navigation patterns and how to develop pattern-based management programs. Section 2 discusses the structure of a pattern-based management program. Section 3 introduces the three main object classes in a management program-- navigation pattern, operator and aggregator. Section 4 gives a concrete example of a management program, using the echo pattern to compute the current load distribution of all links in the network. This example also shows how the migration of control, defined by the pattern, can be decoupled from the management semantics of the program. Finally, section 5 summarizes the results of this paper and gives an outlook on our future work.

2 The Structure of a Pattern-Based Management Program

Figure 2 illustrates the structure of a pattern-based management program. It is comprised of three abstract object classes [14]. The first class, called navigation pattern, specifies how the flow of control of the program migrates from node to node during the course of its execution. The second class, called aggregator, specifies the operation to be performed on each node and how its results are to be aggregated. The third class, called operator, provides a (management) platform independent interface to state and control variables of a network node.

The relationship between the instances of a pattern, an aggregator and an operator is as follows. The navigation pattern controls the execution of the aggregator on the network nodes. The aggregator implements the management semantics of a particular management operation. It runs in an execution environment and accesses node functions through the operator. A navigation pattern can also access the operator to retrieve node information in order to make traversal decisions.

The execution of a pattern-based management program involves asynchronously creating and maintaining a set of distributed states throughout the network. We distinguish among three different abstract classes for these states. The pattern state class contains information used by navigation patterns to traverse the network, while the aggregator state class holds the state of the distributed management operation. Finally, the node state class represent the operational state of a network node that is accessible to management operations.

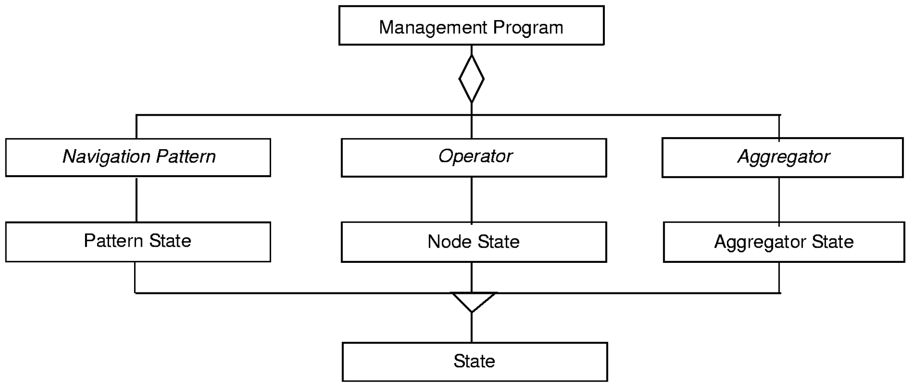


Fig. 2. Class diagram of a pattern-based management program. Abstract classes are in italic.

Pattern and aggregator state can exist in two forms. State that is stored on individual nodes for the lifetime of the management program is called fixed state. Sometimes, it is required that data is carried from node to node as the program executes on the network, in which case we refer to it as mobile state. For example, an aggregator that counts the number of leaf nodes in a network may store sub-totals as fixed states on certain nodes during program execution. In contrast, a navigation pattern that "backtracks" to nodes previously visited may carry its return path in form of a mobile state. While both fixed and mobile states have the same generic structure, the mobile state implements additional functionality, such as data serialization which it requires for state migration.

3 Patterns, Operators, and Aggregators

The abstract class navigation pattern is specialized into concrete classes, each of which defines a particular graph traversal algorithm. These algorithms distribute the flow of control of the management program in a particular manner. For example, the echo pattern shown in the Figure 4 represents a navigation pattern that distributes control according to the wave-propagation algorithm [3][6].

For each concrete class of a navigation pattern, we define a corresponding abstract aggregator class. A pattern and an aggregator must fit together in order to function properly. Therefore, as shown in Figure 3, an abstract echo aggregator class is defined for the echo pattern class. This aggregator class in turn must be specialized into concrete application-specific classes, one for each type of management operation to be implemented. For example, the echo aggregator in Figure 3 is specialized into two different concrete aggregator classes--the leaf counting aggregator, and the aggregator for computing the connectivity distribution of the network topology.

In Figure 3, we also see that the abstract operator class is specialized into classes that interface with specific technologies for node access. In this example, three types of operators are given-- interfaces for SNMP, CMIP and GSMP [10].

As we will show in the following section, we define and implement a navigation pattern using the model of a Finite State Machine (FSM). When a management

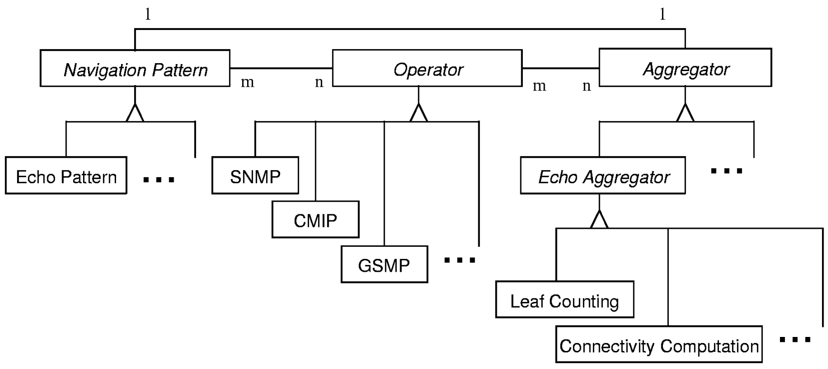


Fig. 3. Specialization of patterns, aggregators and operators. Abstract classes are in italic.

program is launched, control is transferred from the management station to the program's pattern. The execution of a pattern on a network node involves two actions. First, the FSM state of the pattern is determined via the internal logic of the pattern and, second, a function corresponding to that state is triggered in the program's aggregator. Note that for each FSM state, there is exactly one function in the aggregator corresponding to that state. This function defines the local actions or operations to be performed by the aggregator when the pattern is in the associated state. In this sense, the interface between pattern and aggregator can be modelled as an FSM that represent the operation of the pattern and the navigation pattern can be seen as controlling the execution of the management operation (via the aggregator) through those states.

4 Writing a Pattern-Based Management Program

Writing a pattern-based management program is a two-step process. In the first step, a navigation pattern which drives the execution of the management operation, is developed (or selected from a catalogue of well-known patterns). In the second step, an aggregator, which performs the actual management operation, is written. In the following subsections, we discuss both steps in detail.

Consider the development of a management program that obtains the link load statistics and computes the link load distribution across a large network in near real-time. Management operations of this nature provide a network operator with an immediate snapshot about the state of the network and allow decisions to be made in a responsive manner.

The traditional approach to this problem, using a centralized management paradigm, calls for writing a management program that polls every node in the network and then processes the results to compute the histogram. While feasible for small networks, this approach does not scale for most network topologies, due to the increasing processing time and the induced traffic overhead. A different, more sophisticated approach, includes a hierarchy of management entities-- a solution that is not feasible for dynamic networks with frequent changes in topology.

In contrast, a pattern-based management system is distributed in nature and requires no global topology information in order to accomplish the above task. A pattern-based approach to the problem involves selecting a navigation pattern that traverses all nodes in the network and implementing an associated aggregator that computes the required histogram during the pattern's traversal. As we have shown in [6] with the example of running the echo pattern on the Internet, a pattern-based management program can be significantly more efficient in a large network for this task than a centralized system.

4.1 Programming the Echo Pattern

In this section, we show how to implement the echo pattern, a navigation pattern we have introduced in [6]. Management operations based on this pattern do not need knowledge of the network topology, can dynamically adapt to changes in the topology and scale well in very large networks.

The defining characteristic of the echo pattern is its two-phase operation, which works as follows. In the first or expansion phase, the flow of control emanates from the node attached to the management station (start node) via messages called explorers. When an explorer arrives at a node for the first time (unvisited node), it sends out copies of itself on all links, except for the one it arrived from (explorer link). It then marks the node as being visited. When an explorer arrives on an already visited node, it triggers the second phase of the pattern, the contraction phase, by sending a message called an echo out on its explorer link. If an explorer arrives at a leaf node, (one whose only link is an explorer link), it is returned as an echo. When all the explorers of originating from a node have returned as echoes, a new echo is generated and sent out on the node's explorer link. Therefore, for every explorer that is sent over a link, there is a corresponding echo that returns over the same link. The pattern terminates when the last echo returns to the management station.

The execution of echo pattern can be visualized as a wave that propagates out from the start node to every node in the network, before converging back to the start node.

A navigation pattern is implemented as a distributed, asynchronous program that runs identical copies of itself on a set of nodes. On each of these nodes, its execution can be described by an FSM. Upon invocation on a node, the local copy determines its FSM state by examining the fixed and mobile pattern states (Section 2) and triggers a function within the aggregator that corresponds to its current FSM state. When that function returns, it signals to a subset of adjacent nodes (including itself) to schedule its further execution on those nodes.

In the case of the echo pattern, its local program can be described by an FSM with the following seven states:

- ***OnBegin***
  This state is the first state entered when the pattern is launched from the management station. It is triggered exactly once on the start node.
- ***OnTerminate***
  This state corresponds to the termination of the navigation pattern. It is entered on the node where the management program has been launched, after the aggregator triggered by ***OnLastEcho*** state has returned.

- ***OnFirstExplorer***
  This state is entered when an explorer reaches an unvisited node. It is triggered exactly once on each node in the network.
- ***OnSecondaryExplorer***
  This state is entered when an explorer reaches an already visited node.
- ***OnFirstEcho***
  This state is entered when the first echo returns to a node. It is triggered exactly once on each node in the network.
- ***OnSecondaryEcho***
  This state is entered when an echo, other than the last or first echo, arrives at a node.
- ***OnLastEcho***
  This state is entered when the last echo arrives at a node. It is triggered exactly once on each node in the network.

The seven states listed above can be further classified into three broad categories, namely, (a) *initialization and termination states* (consisting of the ***OnBegin*** and ***OnTerminate*** states), (b) *expansion states* (consisting of the ***OnFirstExplorer*** and ***OnSecondaryExplorer*** states) and (c) *contraction states* (consisting of ***OnFirstEcho***, ***OnSecondaryEcho*** and ***OnLastEcho*** states).

The initialization and termination states are used by the aggregator to initialize mobile state variables and to return results back to the management station respectively. The expansion states are typically used to disseminate information, initialize fixed state variables or initiate local operations in aggregators. The contraction states are used to incrementally aggregate results or to propagate the results of operations back to the start node.

Since the echo aggregator specifies the local actions/operations to be performed for each of the possible seven states that the echo pattern can be in at any point in time, it is defined by seven abstract functions corresponding to these states.

Figure 4 shows the implementation of the `run()` method of the echo pattern in C++. Note that the interface definition of this method, restricts access to the operator (`op`), the pattern state (`ps`), the aggregator (`agg`) and the aggregator state (`as`). The program begins by testing if it was just launched by checking the mobile state variable `begin_m` (line 2). If the variable does not exist (meaning the current node is the start node), the program creates it and triggers its aggregator by calling its `OnBegin()` function (lines 3 and 4).

Next, the program checks if it is an echo or an explorer (line 5). If it is an explorer, it checks if the current node has been visited (line 6). If not, the program triggers its aggregator, by calling its `OnFirstExplorer()` function, flags the node as visited and obtains a list of its neighbours from its operator (lines 7 to 10). The program also initializes a fixed state variable `first_echo_f` to indicate that the first echo has not yet returned (line 9).

The program then checks, if a mobile (pattern) state variable `path_m` has been created (line 11). If not, it creates such a variable to store the return path for use during the contraction phase (line 12). If `path_m` has been created before, the program checks, if the current node has any neighbours other than its parent, which would result in further expansion (line 13).(The term ‘parent’ refers to the neighbour node that is connected via the node's explorer link.) If not, the pattern has encountered a leaf node, in which case, the program marks itself as an echo by creating a mobile state variable `is_echo_m` (line 14). It then creates a fixed state variable `num_explorers_f` with value 1 and schedules itself to run on the same node (lines 15 and 16). If the current node is not a leaf node, the program saves its path in `path_m`, creates a fixed state variable `num_explorers_f` to count the number of outstanding explorers, and proceeds by passing control to its neighbour nodes (lines 17 to 19).

If the executing instance of the pattern is an explorer and the current node has already been visited, then the program triggers its aggregator, by calling its `OnSecondaryExplorer()` function (line 20), marks itself as an echo (line 21) and returns to its parent node, whose id is stored in `path_m` (line 22).

If the instance of the pattern program is an echo, it determines if it is the first echo by checking `first_echo_f` (line 23). If so, it triggers its aggregator by calling its `OnFirstEcho()` function and resets `first_echo_f` to indicate that the first echo has returned (lines 24 and 25).

The program then decrements `num_explorers_f` (line 26) and checks the result for zero (line 27). If this is the case, this means that all outstanding explorers have returned as echoes, and the program calls `OnLastEcho()` of its aggregator (line 28). Furthermore, if `path_m` is empty, which would indicate that the pattern has completed its traversal (line 29), the pattern calls `OnTerminate()` of its aggregator (line 30). Otherwise, the pattern calls `OnSecondaryEcho()` of its aggregator (line 32).

As can be seen from the description above, programming patterns can be a non-trivial exercise, requiring a background in distributed algorithms. As such, a key benefit of the pattern-based management approach lies in the re-usability of navigation patterns to solve classes of management problems.

4.2 Programming an Echo Aggregator

Programming the aggregator for computing the link load distribution involves (a) deriving a new class from the abstract class echo aggregator (Figure 3), and (b) defining the semantics of each of its seven functions. Briefly, its operation can be described as follows. While the pattern expands, a histogram is created as a fixed (aggregator) state variable on each node, which is populated with the link utilization of the outgoing links. In the other case, when the pattern contracts, a node's histogram is copied into a mobile (aggregator) state variable and propagated to back its parent in the echo. When the echo arrives at the parent, this histogram is aggregated with the parent's histogram. When all echoes have arrived at the parent node, the parent sends its aggregated histogram to its parent which repeats this process recursively.

```
HopList EchoPattern::run(Operator op, PatternState ps, AggregatorState as,
 Aggregator *agg){

  if (!ps.has_state("begin_m")) {
    ps.create_mobile_state("begin_m", 0);
    agg->OnBegin(as, op);
  }

  if (!ps.has_state("is_echo_m")) {

    if (!ps.has_state("visited_f")) {

      agg->OnFirstExplorer(as, op);
      ps.create_fixed_state("visited_f", 1);
      ps.create_fixed_state("first_echo_f", 0);

      NodeList nl = op.get_neighbors();

      if (!ps.has_stack("path_m"))
        ps.create_fixed_stack_state("path_m", 0)
      else if (nl.num_nodes() == 1) {
           ps.create_mobile_state("is_echo_m", 1);
           ps.set_state("num_echoes_f", 1);
           return make_hop_list(op.node_id());
      }

      ps.push_stack("path_m", op.node_id());
      ps.create_fixed_state("num_explorers_f", nl.num_nodes() - 1);
      return make_hop_list_except_parent(nl, ps.peek_stack("path_m"));

    } else {

      agg->OnSecondaryExplorer(as, op);
      ps.create_mobile_state("is_echo_m", 1);
      return make_hop_list(ps.pop_stack("path_m"));

  } else {

    if (ps.get_state("first_echo_f") == 0) {
      agg->OnFirstEcho(as, op);
      ps.set_state("first_echo_f", 1);
    }

    ps.set_state("num_explorers_f", ps.get_state("num_explorers_f")--);
    if (ps.get_state("num_explorers_f") == 0) {

      agg->OnLastEcho(as, op);
      if (ps.peek_stack("path_m") == 0) {
          agg->OnTerminate(as, op);
          return NULL;
      }

    } else agg->OnEcho(as, op);
  }
}
```

Fig. 4. C++ implementation of the echo pattern

Figure 5 gives the C++ implementation of this aggregator. When the pattern triggers the `OnFirstExplorer()` function of the aggregator (line 2), an empty histogram is created (line 3). The program then calls the operator to read the link utilization statistics of each link, quantizes them and updates the histogram (lines 4 to 6). Finally, it stores the results in a fixed state variable `histogram_f` (line 7). When the `OnSecondaryEcho()` function of the aggregator is triggered (line 10),

```
1  void LinkLoadDistributionAggregator::OnBegin(AggregatorState as,
    Operator op) {
   }

2  void LinkLoadDistributionAggregator::OnFirstExplorer(AggregatorState as, Operator
    op) {
3     histogram *h = new histogram();
4     Links l = op.get_links();
5     for(int i = 0; i < l.count(); i++)
6       h->add(op.get_utilization_quantized(l.link_id(), 1000000), 1);
7     as.create_fixed_state("histogram_f", h);
   }

8   void LinkLoadDistributionAggregator::OnSecondaryExplorer(AggregatorState as,
    Operator op) {
    }

9   void LinkLoadDistributionAggregator::OnFirstEcho(AggregatorState as, Operator op)
    {
    }

10  void LinkLoadDistributionAggregator::OnSecondaryEcho(AggregatorState as, Operator
    op) {
11    histogram *h;
12    if (!as.has_state("histogram_m")) {
13      h = as.get_state("histogram_f");
14      histogram *h2 = as.get_state("histogram_m");
15      h->sum(h2);
16      as.set_state("histogram_f", h);
      }
    }

17  void LinkLoadDistributionAggregator::OnLastEcho(AggregatorState as, Operator op) {
18    histogram *h;
19    if (!as.has_state("histogram_m")) {
20      h = as.get_state("histogram_f");
21      as.create_mobile_state("histogram_m", h);
22    } else {
23      h = as.get_state("histogram_m");
24      histogram *h2 = as.get_state("histogram_f");
25      h->sum(h2);
26      as.set_state("histogram_m", h);
      }
    }

27  void LinkLoadDistributionAggregator::OnTerminate(AggregatorState as, Operator op)
    {
28    histogram *h = as.get_state("histogram_m");
29    cout << "Histogram is " << h.print();
    }
```

Fig. 5. C++ implementation of the link load distribution aggregator

the program checks if a histogram is being returned in the echo through a mobile state variable `histogram_m` (line 12). If so, it aggregates its histogram with the received histogram using the `sum()` method (line 15) and re-saves it (lines 16). Similarly, when the `OnLastEcho()` function is triggered (line 17), the program checks if a histogram is being returned through `histogram_m` (line 19). If not, it creates the histogram from a copy of `histogram_f` (line 21) and returns. Otherwise, it aggregates its histogram with the received histogram (lines 23 to 25) and returns it to its parent as `histogram_m`. Finally, when the `OnTerminate()` function is triggered (line 27), the program prints out the histogram (line 29).

Note that there is generally more than one way to implement an aggregator, namely, by associating aggregator functions with different states of the pattern. For example, in the code of Figure 5, the local operation of reading a node's link utilization statistics is performed in the `OnFirstExplorer()` function, i.e. the function associated with the ***OnFirstExplorer*** state, while the `OnFirstEcho()` function is empty. An alternative implementation for the same operation would be to reverse the associations of these two functions by performing the local operation in the `OnFirstEcho()` function instead. The difference between these two implementations lies in the performance characteristics of the operation. An example showing this difference can be found in [7]. Network managers can take advantage of this effect by choosing implementations according to their performance objectives.

Moreover, although the we distinguished five states--excluding initialization and termination--as part of the echo pattern, one state is sufficient to realize many operations. This state can be understood as combining the three contraction states, namely ***OnFirstEcho***, ***OnSecondaryEcho*** and ***OnLastEcho***, into a single state. Consider the task of computing the average load of all network nodes. An aggregator implementing this task can be written as a function that incrementally aggregates the partial averages delivered via echoes and the local load. In general, functions, such as sum, product, average, min, and max, can be implemented using this approach. These functions are based on operators that are commutative, i.e., , and associative, i.e., .

$$Op(x_i, x_j) = Op(x_j, x_i) \forall(i, j)$$

$$Op(x_i, Op(x_j, x_k)) = Op(Op((x_i, x_j), x_k)) \forall(i, j)$$

As explained above, using more than one state allows the implementor to create operations with different performance profiles, e.g., operations with shorter completion times. For example, if a local read operation takes a long time, it is often more efficient to perform the read operation in one of the expansion states and use the contraction states for aggregation only. Therefore, such an implementation requires (at least) two states, in addition to initialization and termination.

5 Discussion

One of the primary benefits of pattern-based management is that it allows pattern programs to be reused to solve a large class of management problems. This is possible because navigation patterns are generic programs and do not encapsulate the semantics of a management operation. So far, we have successfully designed echo aggregators that compute network load statistics and distributions, calculate global topological properties and perform network initialisation.

In general, aggregators are also significantly less complex than navigation patterns. Patterns are distributed programs that must be designed operate correctly in any given network topology. Aggregators, on the other hand, are local function calls. Aggregator programmers do not need to contend with issues related to network topology, concurrency, synchronization or termination. Therefore, we envision a pattern-based management system to contain a catalogue of a few carefully designed patterns developed by distributed systems specialists and a wide variety of aggregators written by management application programmers.

In our approach, we model the interface between a pattern and an aggregator as a FSM. This design hides the distributed nature of the algorithm implementing the navigation pattern from the programmer of the aggregator. In many cases, the number of states of this state machine can be reduced even further. The cost of this simplification, as we have shown in section 4.2, is that the programmer has less of a choice in selecting specific performance profiles. However, as the example also clearly shows, even this one-state FSM can be used to implement a large class of useful computations, all without requiring any understanding of how the echo pattern works.

In order to support the development of pattern-based management programs and to study their scalability properties, we have built a software workbench for constructing, testing and simulating a pattern-based management system [7]. The tool, called SIMPSON, is written in Visual C++ and runs on Microsoft Windows platforms (Win 95, 98, NT, 2000). Currently, it allows for the interactive simulation and visualization of patterns on networks as large as 10,000 nodes.

We believe that this work opens up interesting avenues for further research. Currently, we focus on the following aspects. First, we want to explore the applicability of our approach to provisioning and management of Internet services, such as DiffServ. Second, we want to analyse navigation patterns in terms of survivability properties. For example, the echo pattern in this paper is not resilient to node or link failures and needs to be enhanced to increase its robustness. Further, we plan to investigate which classes of management problems can be solved in an asynchronous, symmetrical and distributed way, which makes them ideal candidates for implementation on a pattern-based management system. Above all, we want to develop an inventory of navigation patterns that are applicable to key management tasks, analyse them regarding performance, scalability, and survivability, and implement them as software components that can be embedded in management platforms.

Acknowledgments. The work described in this paper has been supported by a grant from British Telecom.

References

[1] M. Baldi and G.P. Picco, “Evaluating the Tradeoffs of Mobile Code Design Paradigms in Network Management Applications,” in Proceedings of ICSE '98, Kyoto, Japan, April 1998, pp. 146-155.

[2] J. Banks (Ed), J. S. Carson, B. L. Nelson and D. M. Nicol, Discrete Event System Simulation, Prentice-Hall, August 2000.

[3] E. J. H. Chang, “Echo Algorithms: Depth Parallel Operations on General Graphs,” IEEE Transactions on Software Engineering., vol. 8, no. 4, pp. 391-401, July 1982.

[4] E. Gamma, R. Helm, Ralph Johnson, and John Vlissides: Design Patterns—Elements of Reusable Object-Oriented Software, Addison Wesley, 1995.

[5] R. Kawamura and R. Stadler: “A middleware architecture for active distributed management of IP networks,” in Proceedings of IEEE/IFIP NOMS 2000, Honolulu, Hawaii, April 2000, pp. 291-304.

[6] K.S. Lim and R. Stadler, “A Navigation Pattern for Scalable Internet Management,” in Proc. of 7th IFIP/IEEE IM’01, Seattle, USA, May 2001, pp. 405-420.

[7] K.S. Lim and R. Stadler, “Developing Pattern-Based Management Programs,” CTR Technical Report 503-01-01, Columbia University, New York, July 2001.

[8] N. Lynch, Distributed Algorithms, Morgan Kaufmann Publishers, August 1997.

[9] R. Marshall, The Simple Book, Prentice Hall, 1994.

[10] P. Newman,W. Edwards, R. Hinden, E. Hoffman, C. F. Liaw, T. Lyon and G. Minshall, “Ipsilon's General Switch Management Protocol Specification Version 2.0,” RFC 2297, March 1998.

[11] V. A. Pham and A. Karmouch, “Mobile Software Agents: An Overiew,” IEEE Communications Magazine, July 1998, Vol. 36 No. 7, pp. 26-37.

[12] D. Raz and Y. Shavitt, “An active network approach for efficient network management,” in Proc. of IWAN'99, (LNCS 1653), Berlin, Germany, June/July 1999, pp. 220-231.

[13] E. Rosen, A. Viswanathan and R. Callon, “Multiprotocol Label Switching Architecture,” RFC 3031, March 1998.

[14] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy and W. Lorenson, Object-Oriented Modeling and Design, Prentice Hall, January 1991.

[15] W. Stallings, SNMP, SNMPv2, SNMPv3, and RMON 1 and 2, Addison-Wesley Publishers, January 1999.

[16] Y. Yemini, G. Goldszmidt, and S. Yemini, “Network Management by Delegation,” in Proceedings of 2nd IFIP/ISINM ‘91, Washington, DC, USA, April 1991.

[17] Y. Zhu, T.M. Chen and S.S. Liu, “Models and Analysis of Trade-offs in Distributed Network Management Approaches,” in Proceedings of 7th IFIP/IEEE IM’01, Seattle, USA, May 2001, pp. 391-404.