Co-Design Patterns for Embedded Network Management

Dominique Dudkowski
NEC Laboratories Europe, Network Research Division
Heidelberg, Germany
dudkowski@nw.neclab.eu

ABSTRACT

Designing and operating large-scale management systems has become extremely challenging due to the growing complexity of network technologies and networked service infrastructures. Both knowledge and functionality for performing management tasks are typically shared between service and management realms. However, current management practices do not adequately address this situation, and management functions are added only after services are deployed. In this paper, we introduce *co-design patterns* to embedded network management and show how they allow for a more structured design of recurring management tasks. Using a distributed fault management scenario we demonstrate how co-design patterns facilitate the interworking of service and management processes that share knowledge and functionality to handle faults collaboratively. We further show that applying co-design patterns results in significant improvement in the runtime performance of embedded management processes.

Categories and Subject Descriptors

C.2.3 [**Computer-Communication Networks**]: Network Operations – *network management.* D.3 [**Programming Languages**]: Language constructs and features – *patterns.*

General Terms

Design, Management, Performance, Languages.

Keywords

Future Internet management, co-design patterns, embedded network management, distributed fault management.

1. INTRODUCTION

Current network and service infrastructures are complex and heterogeneous large-scale deployments, provided by a multitude of stakeholders and deployed dynamically. This situation has made the provisioning of carrier-grade networked systems a challenging task for operations and management. To this day, many solutions are *add-on*, that is, management functions are attached only after networks and services are deployed. Technologies based on SNMP, for instance, provide a generic approach to interface data plane and management plane, but will not be able to cope with future network architectures in the long run, one reason being the separation of management and service plane at operation time, let alone at design time.

In designing complex communication systems, a number of principles have been previously applied with great success, such as modularity, layering, hierarchies, and various forms of interaction (e.g. cross-layering). Complementary, design patterns have emerged to facilitate the implementation of large software systems [1, 2]. Some principles and patterns have received great attention in future Internet research, where new design methodologies are explored that model the complete development cycle from the design to the deployment of whole communication networks [3]. While some principles are common practice today also in network management, such as SNMP's hierarchical management structures, lack of support for the structured design of *embedded management processes* persists. We argue in [4] that integrative aspects of management and service realms should be exploited in the design of management solutions, because in a large number of cases, *both knowledge and functions* for realizing management tasks are shared between both realms. Current principles do not sufficiently assist in this intrinsic design problem, which requires considering design patterns at a finer level of detail.

To this end, we introduce *co-design patterns* to network management that support the design of embedded, distributed, and large-scale management systems. We propose a first set of such patterns (Sec. 2) that we have derived from typical distributed management problems and which are suitable for modeling cooperative aspects of embedded management and service processes. We show by applying a subset of the proposed patterns to a fault management scenario (Sec. 3) how a simple but complete management control loop can be constructed by combining the knowledge and functions required for fault handling from both service and management logic. The scenario illustrates particularly well how a deliberate selection of co-design patterns avoids the duplication of functions on the side of management processes. We further evaluate the scenario analytically (Sec. 4) and show how the application of co-design patterns translates into significant performance gains during runtime, in contrast to a more traditional realization of the same scenario. Finally, we briefly discuss related work (Sec. 5) and conclude in Sec. 6.

2. CO-DESIGN PATTERNS

The concept of co-design originates from the observation that knowledge and functionality about how to manage a system is typically split between multiple roles involved in the operation and management of the target system. For instance, service designers and network operators may each not only know how to manage different aspects of a service, but also how to provide the functions that implement different parts of the overall management tasks. ***Co-design patterns*** proactively support the exploitation of synergies between such parties: they **represent a set of structural blueprints of how to construct parts of a management system by combining knowledge and functionality of different parties** to facilitate the reuse of existing functionality, to simplify management function design, and to increase system performance.

2.1 Embedded Network Management

Co-design patterns are applicable in the context of *embedded network management* where management functions are co-located with service functions. Figure 1 illustrates the concept of *in-network management* (INM), an approach to embedding management functions inside of network elements (network nodes) [4].

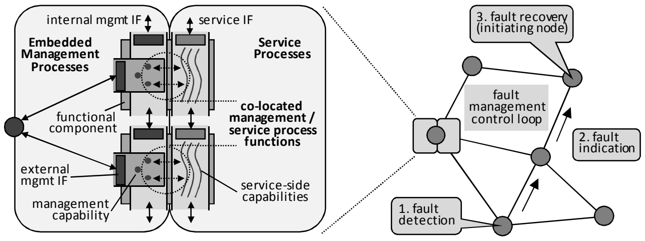
internal mgmt IF
service IF
Embedded Management Processes
Service Processes
functional component
co-located management / service process functions
external mgmt IF
management capability
service-side capabilities
3. fault recovery (initiating node)
fault management control loop
2. fault indication
1. fault detection

Figure 1. INM-compliant node structure (left) and distributed fault management control loop (right).

Figure 1 (left) sketches the structure of a network node, where *functional components* integrate both service and management logic into a single coherent, deployable entity. Management functions are invoked by calls to *management capabilities*, which implement algorithms that realize the management functions, such as fault handling. Typically, a management control loop is formed by multiple interacting management capabilities of functional components that span several nodes in a communication network (Figure 1 (right)). Invocation of management capabilities by service processes and vice versa is performed, for instance, by function calls, and control between both sides is transferred accordingly. Supporting in the design of interactions between embedded management and service processes is the objective of the proposed co-design patterns.

2.2 An Initial Set of Co-Design Patterns

We propose an initial nonexhaustive set of co-design patterns for embedded network management that model typical recurring problems in the fine-granular interactions between management and service functions. For each pattern, a formal representation defines the interactions between service processes S and management processes M. Arrows indicate the process spaces crossed, e.g. S ⇨ M denotes the traversal from service process to management process space. M^+ denotes multiple sequential and M^n multiple concurrent invocations of management capabilities within the management process space.

Control handover (S ⇨ M or M ⇨ S): This basic co-design pattern makes explicit that control is handed between service process and management process in either direction. This pattern separates both spaces in *functional* terms and helps in understanding the separation of concerns in the design phase of complex management systems involving many functional components and network elements. This pattern applies, for example, to the situation of a service-side security exception that leads to the invocation of a security-related management capability.

Informed handover (S ⇨ M or M ⇨ S): This variation of the control handover pattern uses additional information to indicate that the invoked management capability (function) is to be executed with certain constraints. Typically, constraints relate to performance and security, e.g. the maximum delay that only nodes within a network cluster must report on a management-related query. This pattern makes explicit the *knowledge* shared between service and management side, such that both sides can agree on this knowledge and are able to continue concurrent operation with the same assumptions. An example application of this pattern is the invocation of a management capability by a service process with specific timing requirements.

Predicate (S ⇨ M): This pattern provides defined means to evaluate a condition (predicate) on the management side where the knowledge about the condition is provided by the service side. Such situations typically occur in managing faults that can be described by service-specific properties, such as the reception of a sequence of messages that is not allowed in the case of non-faulty operation of the service. Together with a predicate, the information required to evaluate the predicate is handed to the management side for evaluation. By definition, if the predicate evaluates to false, control returns to the service process, otherwise control is resumed on the management side. Hence, the predicate pattern can be viewed as a conditional version of the control handover, and it can be combined with the informed handover. An example application of this pattern is the definition of a fault situation in the form of a predicate that is evaluated by the management side.

Control split (S ⇨ M^n) and *control join* (M^n ⇨ S): These patterns are inspired by multithreading environments. The control split pattern extends the control pattern by transferring control to multiple management capabilities that each execute their own thread of control on the management side. Conversely, the control join pattern defines a synchronization point for multiple control threads executed on the management side such that after the joining of threads, control continues within a single functional component on the service side. While both patterns provide a structured way for creating multiple management control threads, it is the responsibility of the implementation to terminate threads appropriately at a single functional component. Both patterns can be applied, for instance, to execute multiple management capabilities concurrently for the purpose of distributed SLA enforcement, and to synchronize again before returning to the service side.

Control tunnel (S ⇨ M^+ ⇨ S): This pattern enables a service to specify a state that is transferred via a "tunnel" while a sequence of management capabilities are executed on the management side, until it is injected back into the service process that is called by the last management capability in that sequence. This pattern is applicable in situations where service-side functions need the support of intermediary management functions, e.g. to take advantage of more robust or secure mechanisms. As an extension, the state may be modified by the management side to consider decisions that are based on the specific knowledge of a management process. A typical application of this pattern is for a service to select another service's function to be invoked after the traversal of the management side.

3. CO-DESIGN PATTERN APPLICATION

Let us now demonstrate how co-design patterns can be applied in a distributed fault management scenario. Without loss of generality, we consider a mobile ad-hoc network (MANET) that is prone to network partitions. The following arguments are equally valid for other types of networks, e.g. peer-to-peer networks in which a disruption in the overlay may lead to overlay network partitions.

3.1 Distributed Fault Management Scenario

We assume a MANET formed by mobile devices (e.g. in an urban environment) that establish mutual communication links when entering each others' communication range (Figure 2 (left)). Client nodes query a distributed storage service (cf. [5]) to retrieve data items from light-weight data servers. Each server stores information about data items located in the vicinity of a geographic reference point. Appropriate geometric routing protocols ensure that queries are routed via intermediary nodes to the data servers storing the data items requested by client nodes.

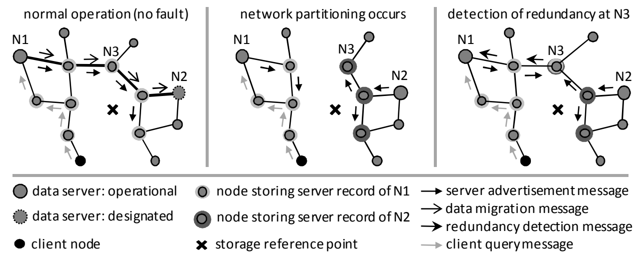
normal operation (no fault)
network partitioning occurs
detection of redundancy at N3
N1
N2
N3
data server: operational
data server: designated
client node
node storing server record of N1
node storing server record of N2
storage reference point
server advertisement message
data migration message
redundancy detection message
client query message

Figure 2. Distributed fault management scenario. *Left:* storage service operation under normal conditions. *Middle:* occurrence of a network partition during data migration. *Right:* joining network partitions and fault detection.

Because data servers are also mobile, a vital part of the service infrastructure is a server advertisement process implemented by servers to advertise their presence in the network so they can be located during query routing. Each advertisement is propagated within a limited scope and deposits a number of server records at intermediary nodes. Due to a server's mobility, it will eventually require to hand over its stored data to an alternative node, which will become the new server. This process of *data migration* (cf. [6]) assures that data is maintained close to the reference point for queries to be always processed efficiently.

Figure 2 (middle) shows the situation where the previously initiated data migration process faces the occurrence of a network partition due to node movement. Communication theory dictates that interrupted migration processes cannot be consistently rolled back in cases where a migration is about to complete but final acknowledgements are pending delivery. As a consequence, two redundant servers remain that handle the same data. In a network where nodes move autonomously, partitions will eventually join, allowing the two servers' data subsets to be merged. In Figure 2 (right), advertisements from both servers, N1 and N2, coincide at N3, a fact that can be exploited in fault detection.

3.2 Scenario Analysis and Co-Design

The described scenario yields the management control loop shown in Figure 1 (right) comprising a fault detection, fault indication, and fault recovery phase. Regarding fault detection, the knowledge that is required to define the fault is on the service side, that is, the advertisement process. Specifically, the reception of advertisement messages from two different servers that are responsible for storing data of the same reference point uniquely characterizes a server redundancy. This suggests to apply the predicate pattern, as illustrated in Figure 3 (left). Incoming messages on the service side are handed over to the management side by an efficient function call with the predicate that specifies the fault and which is evaluated by the invoked management capability. Once the predicate evaluates to true, control flow continues on the management side and on the service side otherwise. While the predicate evaluation logic can be implemented on the service side, the pattern-based solution is more elegant and allows for much more flexibility. For instance, the management capability implementing the predicate evaluation can be reused by other service processes, or even be relocated within the node or to other nodes if this is required in a particular scenario.

In case of a true predicate, control continues on the management side (fault indication). In our scenario, the management side implements a robust protocol that ensures the delivery of the fault indication to a data server. Once the fault indication is received by the server, our implementation supports the parameterization of data migration in such a way that a data merge, which is functionally very similar to a migration, is performed instead of a migration. The control tunnel pattern can be applied in this situation (Figure 3 (right)), because it allows to specify a state that is tunneled through the management side during fault indication. In the example, the state corresponds to an algorithm selection, that is, to perform a data merge. The application of the control tunnel pattern demonstrates how the management side does not have to implement its own recovery mechanism, but can rely on the service side's provided modified migration mechanism.

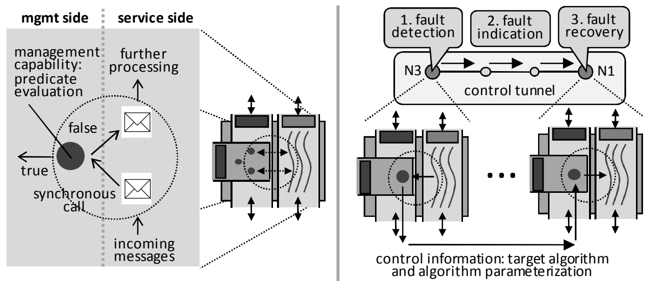

Figure 3. Application of the predicate (left) and control tunnel co-design pattern (right).

The application of both patterns shows that despite the tight integration of service and management processes, it is possible to maintain functional separation that facilitates function reuse. Co-design patterns moreover enable the construction of flexible control loops whose individual algorithm and protocol components can be modified and exchanged when required. For instance, it is possible to adapt the fault indication mechanism on the management side without impacting other parts of the control loop. While we have illustrated the use of two of the proposed design patterns only, our implementation also makes use of concurrent control tunnels that can be modeled using the control split and join pattern, which underpins the power of using multiple co-design patterns in combination.

4. EVALUATION

To demonstrate the performance gains of our co-designed embedded management solution we combine simulation-based and analytical modeling: Using the ns-2 simulator, we have implemented a complete data migration suite in MANETs from which we obtain basic MANET-specific performance characteristics. We then use an analytical model to validate the performance of the co-designed solution. The simulation scenario extends over a geometric region of $600 \cdot 600$ m^2, populated with 150 nodes each with a communication range of 100 m. Nodes move with a constant speed of 15 m/s and fixed pause time of 30 s according to the random waypoint mobility model. The geometric region is subdivided into cells of $200 \cdot 200$ m^2, a total of 9 data servers each stores 320 kB of data to be migrated when a server leaves its associated cell.

Table 1 summarizes basic numerical results that we have extracted from simulations of the above scenario and which we require for the subsequent analyses. The migration failure probability specifies with which probability an initiated migration fails and leads to a server redundancy. The migration duration is the total time from initiating a migration at the source data server to completing it at the designated server. The one-hop packet delay represents a typical delay that a packet takes between adjacent nodes, including all contributions such as transmission and queuing delays. The detector-server hop distance is a representative number of hops that is required for sending a fault indication from the fault-detecting node to the fault-handling data server. Finally, the migration cost are the approximate number of packets required to transfer a server's stored data to the designated one.

Parameter and Symbol	Magnitude (typical)
Migration failure probability p_{fail}	1/121
Migration duration Δt_{mig}	2.65 seconds
One-hop packet delay Δt_{hop}	3.5 ms
Detector-server hop distance n_{hops}	3
Migration cost N_{mig}	$\approx$ 600 (default)

Table 1. Basic simulation-based numerical results.

The total fault recovery time, $\Delta t_{\text{recovery}}$, is the time required to process the complete control loop as illustrated in Figure 3 (right) and takes the general form

$$\Delta t_{\text{recovery}} = \Delta t_{\text{detection}} + \Delta t_{\text{indication}} + \Delta t_{\text{handling}} \qquad (1)$$

In the *co-designed* solution, a fault is detected after the duration of the involved network partition, Δt_{part}, and the time required for concurrently sending advertisement messages from two servers to a detecting node, hence $\Delta t_{\text{detection}} = \Delta t_{\text{part}} + n_{\text{hops}} \cdot \Delta t_{\text{hop}}$. Value Δt_{part} depends on the movement of network nodes and we will use it as a dynamic performance parameter in the evaluation later on. A fault indication is propagated the same distance in inverse direction and requires time $\Delta t_{\text{indication}} = n_{\text{hops}} \cdot \Delta t_{\text{hop}}$. The duration of fault handling is identical to the migration duration (cf. Table 2). The total fault recovery time, $\Delta t^{\text{co}}_{\text{recovery}}$, as a function of partition duration is:

$$\Delta t^{\text{co}}_{\text{recovery}} \approx \Delta t_{\text{part}} + \Delta t_{\text{mig}} + 21 \text{ ms} \qquad (2)$$

In the *non-co-designed* solution, no synergies between service and management processes are exploited. We assume in this case a single stationary management node located in the center of the scenario region that queries the status of all reachable nodes in the network in time intervals Δt_{check} by a flooding-based broadcast. Server replies are aggregated by the management node, which in turn determine pairs of redundant servers. As in the co-designed solution, redundant servers are reachable and the fault can be handled right away. Because a fault can be detected for the first time only after partitions have joined again, fault detection requires at least Δt_{part}. Since a check is performed only every Δt_{check}, an additional $0.5 \cdot \Delta t_{\text{check}}$ is added to the fault detection time. Further, the duration of the aggregation phase depends on twice the network radius, which is $\sqrt{n}$. With the one-hop delay Δt_{hop}, the mean fault detection time is $\Delta t_{\text{detection}} = \Delta t_{\text{part}} + 0.5 \cdot \Delta t_{\text{check}} + \sqrt{n} \cdot \Delta t_{\text{hop}}$.

A fault indication crosses half the radius of the network in the mean, hence $\Delta t_{\text{indication}} = 0.25 \cdot \sqrt{n} \cdot \Delta t_{\text{hop}}$. Finally, fault handling time is identical to the duration of the migration process in the co-designed solution. The total fault recovery time, $\Delta t^{\text{non-co}}_{\text{recovery}}$, sums up to:

$$\Delta t^{\text{non-co}}_{\text{recovery}} \approx \Delta t_{\text{part}} + 0.5\Delta t_{\text{check}} + 4.37\sqrt{n}\text{ ms} + \Delta t_{\text{mig}} \quad (3)$$

We further consider the communication cost for executing the management control loop. Because the co-designed and non-co-designed solution use different schemes, it is convenient to quantify cost, C, in number of packets per unit time. For the control loop we can state:

$$C_{\text{recovery}} = C_{\text{detection}} + C_{\text{indication}} + C_{\text{handling}} \quad (4)$$

Let f_{mig} denote the migration frequency, that is, the number of migrations occurring in the scenario every second. With the migration failure probability from Table 2, the expected failures per second are $f_{\text{mig}} \cdot p_{\text{fail}}$.

In the *co-designed* solution, advertisement messages received by each node are part of the service side, hence, they do not incur additional management overhead and thus, $C_{\text{detection}} = 0$. Fault indication requires the traversal of $n_{\text{hops}} = 3$ hops and fault handling is equal to the migration cost N_{mig} (cf. Table 2). Hence:

$$C^{\text{co}}_{\text{recovery}} \approx (n_{\text{hops}} + N_{\text{mig}}) \cdot f_{\text{mig}} \cdot p_{\text{fail}} \quad (5)$$

In the *non-co-designed* solution, fault detection requires the utilization of explicit messages because no "free" messages on the service side can be exploited. After each Δt_{check}, the aggregation phase requires n flooding packets. Additionally, up to N servers each respond with a mean communication cost that corresponds to half the radius of the network, that is, $0.25\sqrt{n}$. Hence, the total cost is $0.25\,N \cdot \sqrt{n}$, and the fault detection packet rate is $C_{\text{detection}} = (n + 0.25\,N \cdot \sqrt{n}) / \Delta t_{\text{check}}$. Each detected fault is handled by sending messages concurrently from the management node to all redundant servers. This requires approximately the mean distance from the management station to half the perimeter of the region, that is, $0.25\sqrt{n}$ hops. Because it does not matter whether faults are handled in a batch after Δt_{check} or distributed equally over time as in the co-designed solution, the fault indication cost can be stated right away as $C_{\text{indication}} = 0.25\sqrt{n} \cdot f_{\text{mig}} \cdot p_{\text{fail}}$. Similarly, fault handling cost are aggregated instead of equally distributed, and become $C_{\text{handling}} \approx N_{\text{mig}} \cdot f_{\text{mig}} \cdot p_{\text{fail}}$. The total recovery cost rate for the non-co-designed case sum up to:

$$\begin{aligned} C^{\text{non-co}}_{\text{recovery}} \approx\ & (n + 0.25\,N \cdot \sqrt{n}) / \Delta t_{\text{check}} + \\ & 0.25\sqrt{n} \cdot f_{\text{mig}} \cdot p_{\text{fail}} + N_{\text{mig}} \cdot f_{\text{mig}} \cdot p_{\text{fail}} \end{aligned} \quad (6)$$

Figure 4 and 5 visualize the derived equations. The mean fault recovery time is shown in Figure 4 as a function of the checking interval. Three partition duration intervals are shown, f_{mig} is set to 1/s for all graphs displayed. The co-designed solution clearly outperforms the non-co-designed one in every case.

Figure 5 presents three graphs that show the communication cost required for fault recovery for three different settings of the number of migration packets. In all cases, the co-designed solution has again superior performance over the non-co-designed solution. What's more, choosing small checking intervals is not supported by this figure due to the significant increase in communication cost (note the logarithmic scale on the y-axis). Hence, Figure 4 and 5 make clear that the non-co-designed solution is not adequate in either case, whereas the co-designed solution performs efficient and even constant in the considered settings.

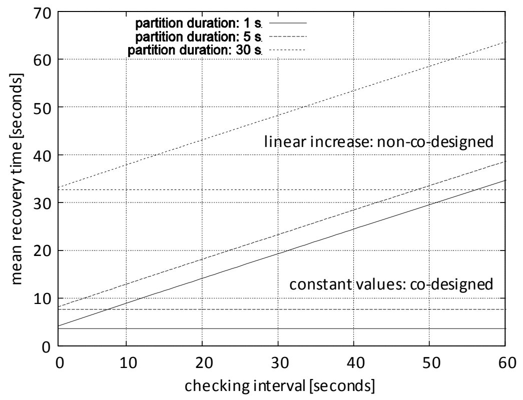

Figure 4. Mean fault recovery time in seconds.

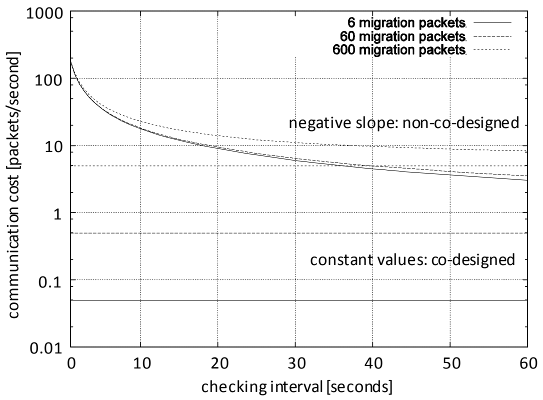

Figure 5. Communication cost in packets/second.

Besides the presented results, the co-designed solution brings the additional advantage of load-balancing fault recovery homogeneously over time, while the non-co-designed approach handles faults in bursts, which is an undesired behavior especially in wireless networks where congestions are more likely to occur.

5. RELATED WORK

Since their first introduction in [1], design patterns have been successfully applied in diverse fields of software design in general (e.g. [2]) and in communications software in particular (e.g. [7]). Patterns in OSS that build mostly on existing ones from distributed systems were recently discussed in [8].

Besides these general treatises, a number of design patterns for sensor networks were identified e.g. in [9], modeling the sharing of results and certain forms of interaction. In context-aware computing, a number of patterns were proposed in [10] to support in the design of context-aware adaptation processes. Recognizing a lack of structure in the design of secure VoIP solutions, the authors of [11] introduce several patterns that address VoIP security problems, such as secure firewall traversal.

In telecommunications system architecture, a number of patterns were introduced in [12] and [13], including the mediator and observer pattern. Further, [14] introduces a number of patterns for managing communication networks, including the manager-agent and managed object pattern. The authors of [15] identify some of the existing standard design patterns in SNMP-based network management, such as the façade pattern, in order to advocate the strong points of SNMP-based management for future management architectures. Finally, the authors of [16] introduce the echo pattern that facilities the execution of distributed management operations by a two-phase distributed dissemination and aggregation protocol.

In conclusion, the large collection of design patterns considered in the literature range from established patterns in software engineering (e.g. façade pattern) to very specific patterns in network management (e.g. echo pattern). However, the proposed patterns are not applicable to the design of management tasks where both knowledge and functionality are shared between service and management processes that have the potential to cooperate in accomplishing complex management operations.

6. CONCLUSION

We have described co-design patterns that provide a structured way of implementing management functions where service and management processes share knowledge and functionality of how to accomplish a management task. By using the example of distributed fault management, we have shown how co-design patterns can be applied to simplify design and exploit existing functionality and how the application of co-design patterns yields significant performance benefits.

While we have identified a first set of valuable co-design patterns, more experience is required to conceive and evaluate the benefit of other potential such patterns. Currently, we are pursuing this goal in the context of the in-network management paradigm within the 4WARD project [4]. Of specific interest is how the co-design patterns can be used to facilitate the transition of current management practices towards future ones, that is, how to move from centralized to more distributed management architectures. While co-design patterns will not solve the full spectrum of complexity problems in distributed management systems, they can contribute significantly to supporting simplicity, reusability, and the distribution of management tasks.

7. REFERENCES

[1] Alexander, C.: A Pattern Language: Towns, Buildings, Construction. Oxford University Press (1977)

[2] Gamma, E., Helm, R., Johnson, R., Vlissides, J. M.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley (1994)

[3] Johnsson, M., Huusko, J., Frantti, T., Andersen, F.-U., Nguyen, T.-M.-T., de Leon, M. P.: Towards a New Architectural Framework – The Nth Stratum Concept. MobiMedia'08. Oulu, Finland (2008)

[4] Dudkowski, D., Brunner, M., Nunzi, G., Mingardi, C., Foley, C., Ponce de Leon, M., Meirosu, C., Engberg, S.: Architectural Principles and Elements of In-Network Management. Mini-Conference IM'09. Long Island, NY, USA (2009)

[5] Dudkowski, D, Marrón, P. J., Rothermel, K.: An Efficient Resilience Mechanism for Data Centric Storage in Mobile Ad Hoc Networks. MDM'06. Nara, Japan (2006)

[6] Dudkowski, D., Marrón, P. J., Rothermel, K.: Migration Policies for Location-Centric Storage in Mobile Ad-Hoc Networks. MSN'07. Beijing, China (2007)

[7] Rising, L., Schmidt, D. C.: Design Patterns in Communications Software. Cambridge University Press (2001)

[8] Ashford, C., Gauthier, P.: OSS Design Patterns: A Pattern Approach to the Design of OSS Interfaces. Springer (2009)

[9] Tei, K., Fukazawa, Y., Honiden, S.: Applying Design Patterns to Wireless Sensor Network Programming. ICCCN'07, pp. 1099-1104. Honolulu, HI, USA (2007)

[10] Gordillo, S., Rossi, G., Lyardet, F.: Design Patterns for Context-Aware Adaptation. Saint 2005 Workshop on Context-Aware Adaptation for the Mobile Internet, pp. 170-173. IEEE Computer Society (2005)

[11] Anwar, Z., Yurcik, W., Johnson, R. E., Hafiz, M., Campbell, R. H.: Multiple Design Patterns for Voice over IP (VoIP) Security. IPCCC'06, pp. 492-499. Phoenix, AZ, USA (2006)

[12] Duell, M.: Managing Change with Patterns. IEEE Communications Magazine 37(4):37-38 (1999).

[13] Meszaros, G.: Design Patterns in Telecommunications System Architecture. IEEE Communications Magazine 37(4):40-45 (1999).

[14] Keller, R. K., Tessier, J., von Bochmann, G.: A Pattern System for Network Management Interfaces. Communications of the ACM 41(9):86-93. New York, NY, USA (1998)

[15] Sevinç, P. E., Martin-Flatin, J.-P., Guerraoui, R.: Patterns in SNMP-Based Network management. ICSSEA'04, Paris, France (2004)

[16] Lim, K. S., Stadler, R.: A Navigation Pattern for Scalable Internet Management. IM'01, pp. 405-420, Seattle, WA, USA (2001)