GSpace: An Architectural Approach for Self-Managing Extra-Functional Concerns

Giovanni Russello
Department of Computing
Imperial College London
g.russello@imperial.ac.uk

Naranker Dulay
Department of Computing
Imperial College London
n.dulay@imperial.ac.uk

Michel Chaudron
LIACS
Universiteit Leiden
chaudron@liacs.nl

Maarten van Steen
Dept. of Computer Science
Vrije Universiteit
steen@cs.vu.nl

ABSTRACT

Middleware-based solutions for self-managing systems provide a degree of separation between the mechanisms that govern the adaptability of a system and application functionality. Systems become in this way more flexible, dependable and robust to changes. However, it is possible to achieve another degree of separation by separating from the application logic the different extra-functional concerns (such as availability, performance, and security). This separation, known as Separation of Concerns principle, helps in generating software artifacts that are more maintainable and reusable.

In this paper, we propose an architectural model for a middleware-based solution where the self-managing principle is applied to extra-functional concerns. Our middleware, based on the Shared Data Space model, is capable of dynamically adapt extra-functional concerns to the actual needs of the applications.

1. INTRODUCTION

There is a general consensus in the research community that *self-managing systems* enable software engineers generating applications that are more flexible, dependable and robust to changes in the environment. A self-managing system provides means to adapt the application to changes in the environments and requirements with minimal human intervention. Several proposed approaches indicate that viable solutions for achieving self-adaptation are those where self-managing mechanisms are separated from the application functionality. In this context, middleware-based solutions have proven to be suitable for achieving such separation.

Another degree of separation is that suggested by the *Separation of Concerns* (SoC) principle [2, 9]. According to SoC, the application logic should be separated from concerns that are not directly involved with its basic functionality. Performance, availability and security represent typical *extra-functional concerns* that should be isolated and externalised from the application functionality. Applying the SoC principle positively affects the following properties of applications:

- Application evolution: because the units of abstraction match with the different concerns changing one specific concern does not affect other modules.
- Reusability of concerns: as the code dealing with the application logic can be reused, the same holds for code that deals with extra-functional concerns.
- Concern traceability: traceability among different concerns is increased since each concern is clearly separated from the others.

As discussed by Filman et al. in [3], the SoC principle is essential for developing applications that strongly depend on their environment and that need to continuously evolve and adapt in the course of time. In this paper, we introduce an architecture for a middleware-based approach in which the evolution and adaptation of extra-functional concerns are self-managed. In particular, in our approach application components implement the basic functionality of an application. Our middleware provides the composition abstraction for gluing together application components. Moreover, it provides the necessary mechanisms for self-managing extra-functional concerns.

In line with the adaptation methodology discussed in [8], our middleware supports the management of the evolution and adaptation of extra-functional concerns. Our middleware architecture is based on the shared data space model introduced with the coordination language Linda [4]. The main reason for choosing the shared data space model comes from the fact that this model supports naturally the separation of the computational part of an application form its coordination part. The architecture that we propose in this paper extends the basic model for supporting several extra-functional concerns transparently to the applications. Cur-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
CSTST 2008 October 27-31, 2008, Cergy-Pontoise, France
Copyright 2008 ACM 978-1-60558-046-3/08/0003 ...$5.00.

rently, our middleware supports the separation from the application functionality of the following extra-functional concerns: performance, availability and security. For each of these extra-functional concerns, the middleware is able to enforce specific strategies and adapt these strategies to the actual requirements of the applications.

The contributions of this paper are threefold. First, we propose a specific architecture for separating application functionality from extra-functional concerns. Each extra-functional concern is catered by a set of mechanisms that is available in the framework. New mechanisms can be deployed, extending the life cycle of the framework. Second, our architecture supports modularisation of concern specifications. Functional and extra-functional concerns can be specified in complete isolation to each other. This allows specialists to work on mechanisms specific for a concern without having to know too many details of other concerns. As a consequence, such concern specifications are encapsulated in well-defined modules enhancing the reusability of software artifacts. Not only application code can be reused, but also the concern mechanisms represent reusable units. System architects need not to re-invent mechanisms when the system is deployed in other similar environments. Finally, by separating application functionality from extra-functional concerns, it becomes possible to provide a *dynamic adaptation* of the mechanisms that cater for those concerns during run-time. The architecture offers a feedback loop that monitors the application's behaviour. Whenever the application's behaviour changes the feedback loop adapts the mechanisms to the current needs of the application.

This paper is organised as follows. Section 2, we provide an overview of related research. In Section 3 we list and motivate the requirements that our multi-concern architecture has to satisfy. Section 4 describes our architecture for combining and adapting extra-functional concerns. Section 5 focuses on the interactions that could happen among different concerns and describes methods to deal with such interactions. We give our conclusions and highlight future research questions in Section 6

2. RELATED WORK

In this section, we discuss related work focused on providing the application developer approaches for self-adaptive systems. In line with our approach, work proposed by other research efforts facilitate the separation of certain concerns avoiding that application developers have to entangle their application code with code for managing extra-functional concerns and adaptation related concerns.

Our approach can be considered as a *reflective system.* A reflective system is one that performs computation about itself, and that provides inspection and control through its reflective interfaces [7]. This principle can be applied to middleware. For example, in the OpenCOM component model [1], information about structural and behavioural of components can be accessed through an API. However, this information can be gathered only for a running instance of a component.

In [6], Kramer and Magee propose a general architectural approach providing a three-layers model of a system that is used to evaluate and adapt the running system. In line with this, the Rainbow project [5] introduces an architectural model that extends architecture-based adaptation by adopting architectural styles. Architectural styles capture structural and semantic characteristics common to a family of system. Developers can fine tune architectural styles to tailor specific needs of a system. The Rainbow infrastructure does satisfy the goal of separating adaptation related concerns from application code as discussed above to a certain degree. This approach allows reuse of adaptation properties, mechanisms and strategies between systems that have architectural commonalities that can be expressed in shared architectural styles.

One of the most stringent problems in self-adaptability is deciding the strategy to adapt a running system to better fit the current state of the operational environment. There are several approaches that can be used for different adaptation policies. As discussed in [14], these approaches can be categorised in three different types: (i) event-condition-action (ECA) policies, (ii) goal policies and (iii) utility functions.

ECA policies are adopted in different domains such as computer networks, active databases and expert systems. An ECA policy consists of a rule that specifies exactly what action to take when certain events happened and a condition is true. Although ECA policies are powerful tools for controlling the adaptation of a system, the specification of such rules require a detailed knowledge of the system. Goal polices are more general inasmuch as allow the specification of higher-level performance objectives for an adaptive system. It is up to the adaptation system or middleware to determine the strategy to achieve those objectives. For instance, a typical domain where this type of policies is applied is that of guaranteing promised levels of QoS. Goal policies provide a binary answer to adaptation alternatives in terms of “feasible” and “unfeasible” configurations. On the other hand, utility functions are able to support specifications of “degree of desirability” of different adaptation alternatives of an adaptable service. Utilities are often defined as a function that weighs several metrics of a system in order to select the “best” alternative from a given set.

This last approach is the one adopted in our system. In the following section, we present the requirements that our architecture has to satisfy.

3. REQUIREMENTS FOR A MULTI-CONCERN ARCHITECTURE

In this section, we list and discuss the requirements that driven the design of our architecture. The requirements can be listed as follows:

1. Separation of Concerns: the main goal of our research is that of separating application functionality from extra-functional concerns. Applying this principle increases the modularisation of code and therefore increases the quality of software.

2. Multi-Concern Adaptability: the middleware architecture must support the adaptation of multiple concerns. The application developer can set the goals for her application. The goals specify the level of availability, performance and security. The middleware has to guarantee that such goals are met during runtime even when the environment conditions changes.

3. Transparency of Interweaving: the complexity of the interweaving process for dealing with multiple concerns must be shielded from the application level. In

this way, application developers concentrate on the design of their applications without caring about blending functionality with other concerns.

4. Extendable Set of Strategies per extra-functional concern: adding or modifying strategies for a specific concern must not affect strategies of other concerns.

The first requirement is the foundations of our research. According to SoC, all relevant concerns of a software system should be treated as separated modules. Current approaches provide separation to a certain level, together with decomposition and composition mechanisms. However, these approaches provide a single dimension of separation, with a limited set of tools for decomposition and composition. According to [13], the improvement of quality of software artifacts, the reduction of software production costs, and facilitation of software evolution and maintainability can be fulfilled by realising a full Separation of Concern.

The second requirement focuses on the realisation of a method that automatically combines different strategies that satisfy the requirements of each concern and that minimises the overall costs. Naturally, strategies for different concerns are going to influence each other during the execution of operations. For instance, encrypting and decrypting tuples during communications for security affects the latency for the execution of an operation, impacting the performance of a distribution policy. Realising such a method requires a mechanism for evaluating policies such that:

- it is aware of the interactions between policies for different concerns,
- it should compute a solution in reasonable time.

The third requirement is more concerned about the design of the architecture. For instance, we could design an architecture where a single strategy takes care of availability, performance, and security at once. The code of such a policy would be intertwined with details about different concerns that may turn the writing of such a policy into a nightmare for the developer. Although such a policy is specified outside the code of the application, it violates the SoC principle.

An alternative design choice that is more in line with the SoC principle would be the following. Each strategy should deal with a specific concern. However, each supported concern should have in the architecture a *specific concern subsystem*. In each specific concern subsystem, a *concern manager* decides which strategy should be applied. Now a developer who wants to design a new policy for security, for instance, does not have to care about performance and availability concerns.

However, such a design introduces the issue of how to deal with the interactions between strategies for different concerns. As we will see later, these interactions can be of two different natures and need to be addressed in different ways.

Although we believe that our architecture covers an extensive set of extra-functional concerns for distributed systems, it might be the case that in the future a new extra-functional concern must be supported. This would require the introduction of the specific concern-subsystem and possibly some modifications to accommodate the dependencies introduced by the new concern. However, the design of the architecture guarantees that there is no need for changing any of other concern-subsystems already present in the architecture.

In the next section, we introduce our architecture and more details on its implementation.

4. GSPACE ARCHITECTURE

GSpace is a prototype implementation of our architecture. In GSpace, extra-functional concerns can be treated separately from the application functionality. In previous work, we discussed how GSpace architecture was used for dealing with a single extra-functional concern. In particular, GSpace was used for separating from the application level concerns such as performance [10], availability [11] and security[12]. The results that we obtained show that extra-functional concerns can successfully be treated orthogonally with respect to application functionality. Additionally, applying the SoC principle allows us to implement an architecture where dynamic adaptation is possible. By dynamically adapting a specific concern to the actual requirements, our middleware is able to increase the overall performance of the system. Naturally, the question that arises is: *what happens when multiple extra-functional concerns have to be dealt with at the same time?*

Clearly, when dealing with multiple extra-functional concerns at the same time, interactions between separated concerns may occur. The nature of such interactions may be different depending on which concerns are interacting. In some cases, these interactions could lead to sub-optimal utilisations of the resources of the system. In other cases, conflicts may arise that require a more careful handling.

In this section, we briefly introduce the basic concepts behind the design of GSpace. Following, we describe in more details GSpace architecture for dealing with multiple concerns.

4.1 Basic Concepts

GSpace is an implementation of a distributed Shared Data Space (SDS). The SDS concept was introduced in the coordination language Linda [4]. In Linda, applications communicate by inserting and retrieving data through a data space. The unit of data in the data space is called *tuple*. Tuples are retrieved from the data space by means of *templates*, using an associative method. Multiple instances of the same tuple item can co-exist. An application interacts with the data space using three simple operations: `put`, `read` and `take`.

A typical setup of GSpace consists of several *GSpace kernels* instantiated on several networked nodes. Each kernel provides facilities for storing tuples locally, and for discovering and communicating with other kernels. GSpace kernels collaborate with each other to provide to the application components a unified view of the shared data space. Thus the physical distribution of the shared data space across several nodes is transparent to the application components, preserving its simple coordination model. In GSpace tuples are typed. This allows the system to associate different strategies for extra-functional concerns with different tuple types.

Figure 1 shows a typical GSpace set-up, where several application components and a GSpace kernel are deployed on a networked node. A GSpace kernel consists of two subsystems: the *Operation Processing Subsystem (OPS)* and the *Adaptation Subsystem (AS)*.

The OPS provides the core functionality necessary for a node to participate in a distributed GSpace: handling ap-

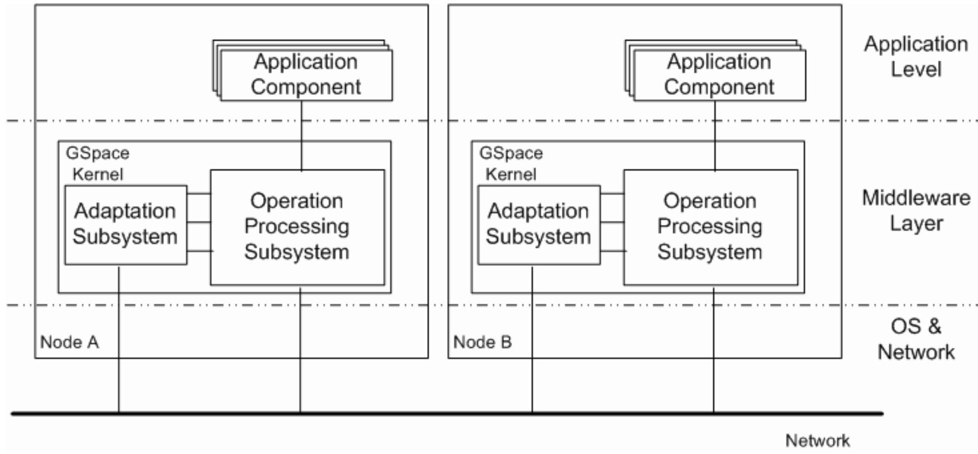

Figure 1: Typical deployment of GSpace Kernels in several networked nodes.

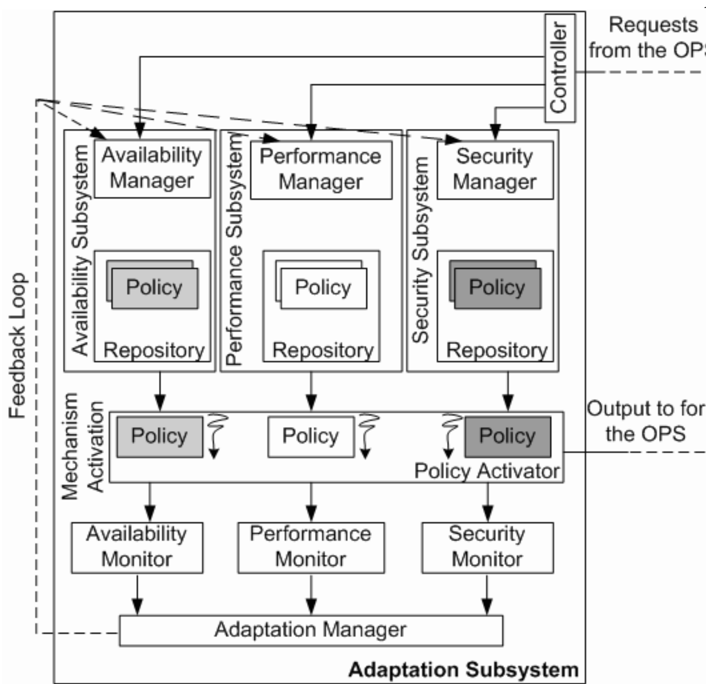

Figure 2: Adaptation Submodule for a multi-concern architecture.

plication component operations; providing mechanisms for communication with kernels on other nodes; and monitoring connectivity of other GSpace nodes that join and leave the system; and maintaining the information about other kernels. Finally, the OPS provides the infrastructure to differentiate distribution strategies per tuple type. The internal structure of the OPS is described in [10]. The AS is described in the following section.

4.2 Self-Managing Extra-Functional Concerns

Figure 2 sketches the design of the AS architecture for self-managing multiple concerns. In the following each of the modules that compose the AS is described.

- **Controller**. This module receives the requests from OPS. The requests represent operation for the SDS that are received from the application level. In order to fulfill these requests, extra-functional concerns have to be applied. Requests are dispatched to the managers of each concern-manager.
- **Concern Subsystem**. For each concern that our framework supports there is a dedicated subsystem in the architecture. Each subsystem contains the following modules:
  - **Concern Manager**. A concern manager has to identify which policy has to be activated in function of the type of operation and type of tuple that the request is carrying.
  - **Policy Repository**. A repository contains a set of policies dedicated to a specific concern. In our approach, strategies and mechanisms for concerns are implemented as *policies*. For each concern, repositories receive directives from the respective managers for activating policies. For instance, for the availability concerns several strategies for replicating the tuple are embedded in availability policies.
- **Policy Activator**. Once the concern managed identifies which policy should be activated for a request, the policy is taken from the repository and activated through the Policy Activator. For each concern, a policy is activated. This means that the policy activator combines the output of the different policy passing the result back to the OPS that completes the application request.
- **Monitors**. For each concern, there is a monitor that collects information on the execution of each policy. Each monitor has to be considered as an array of sensor dedicated to the collection of data specific for each concern. For instance, an availability monitor collects information about the up-time of a node.

- **Adaptation Manager**. This module is responsible for collecting the data from the monitors and combine them to get a quantification of the system performance. This quantification is achieved by means of a *general cost function* (CF_G). More details on the CF_G are given in the next section. Here we can say that such a function is a linear combination of several metrics that capture the trade-off between the cost incurred by the system and its actual performance. Once the CF_G provides the optimal combination of policies for availability and performance, the Adaptation Manager has to check whether such combination is in conflict with the security constrains. After possible conflicts have been solved, the Adaptation Manager informs the managers about the new combination that has to be activated. The managers then switch to the required policies.

Concluding, the calculation of the CF_G is done per tuple type.

5. CONCERN INTERACTIONS

In this section, we discuss how different concerns interact with each other. In particular, we want to analyse the different types of interaction and how we can deal with such interactions in our architecture.

In our architecture, each concern is dealt with in isolation with respect to the others. From the specification of the policy that has to be used to the implementation of the specific mechanism that implements such a policy, concerns are orthogonal to each others. Ideally, such orthogonality should be maintained during execution. In reality, when a policy for one concern is executed it invariably interacts with the execution of other policies.

In the following, we analyse the possible interactions between availability, performance, and security. We first concentrate on the interactions between availability and performance. Afterward, we concentrate on how availability and security interact.

5.1 Availability and Performance Interactions

Our framework allows the specification of policies for availability and performance concerns in isolation to each other (and both in isolation to application functionality). This has the advantage for developers to be able to modularise the development of code for availability and for performance in well-specified and separated units. This allows the developers to concentrate on problems related to concerns once at a time, increasing the understandability and efficiency of their code.

A policy for a concern specifies which mechanism should be activated per tuple type. For instance, in the case of availability, mechanisms are distribution strategies that replicate tuples on several nodes for statistically guaranteing that tuples are still available even in the case that a node crashes. In the case of performance, distribution strategies are used for strategically placing tuples to reduce application latency.

Although the framework separates availability and performance concerns, both concerns leverage on the same type of mechanisms, that is, distribution strategies for tuples. To avoid introducing dependencies between concerns inside the framework (to meet our requirements 1 and 4), mechanisms for different concerns are not aware of each others. This isolation could lead to a sub-optimal usage of resources. For instance, when a `put` operation is executed a mechanism for availability and a mechanism for performance are activated. These mechanisms could request the same set of nodes to store the same tuple instance twice. It should be noted that this interaction of mechanisms does not violate the semantics of the SDS operation. When a `take` operation is executed, both mechanisms will be activated removing both tuple instances. As for `read` operations, the presence of two instances of the same tuple does not affect the semantics of the operation.

However, this modus operandi of the mechanisms has an effect on the resource usage. Inserting the same instance of a tuple twice on the same nodes requires twice as much memory and bandwidth usage without any further increasing availability or performance.

One possible way to handle such situations is the following. Let us consider a combination of two policies (one for availability and one for performance) that results in a non-optimal usage of resources as a *sub-optimal combination*. This combination is such that the constraints for availability and performance are satisfied without optimisation of resource usage. Our problem is shifted to finding a combination of policies such that those constraints are still satisfied and the resource usage is minimised. We refer to such a combination as the *optimal combination*.

To find the optimal combination we resort to our adaptation mechanism based on utility functions, that we called *cost function*. By using a cost function that combines relevant metrics, our adaptation mechanism is able to find the "best" policy among the ones in the repository. In this case, we are interested in evaluating combinations of availability and performance policies.

Let $c(a, p)$ be a combination of availability policy a and performance policy p. To evaluate such a combination we propose the use of a *general* cost function, indicated as CF_G, defined as follows:

$$CF_G(c(a,p)) \;=\; w_1 * CF_A(a) + w_2 * CF_P(p) \qquad (1)$$

where CF_A and CF_P are the cost functions for availability and performance, respectively.

The availability cost function CF_A is defined using the following metrics: bu represents the network bandwidth usage; mu represents the memory consumption for storing the tuples in each local data space; and da represents the *derived availability*. The latter is calculated as follows:

$$da_a = \begin{cases} 100 - avail(a) & \text{if } avail(a) \geq required_avail \\ MaxValue & \text{if } avail(a) < required_avail \end{cases}$$

In this way, if the availability provided by an availability policy a does not satisfy the availability required by the user then the value for da is set to $MaxValue$ (with $MaxValue \geq 100$) so that the calculated costs will become very high and the system will automatically reject this policy. The cost function is defined as follows:

$$CF_A(a) \;=\; w'_1 * bu_a + w'_2 * mu_a + w'_3 * da_a \qquad (2)$$

For the performance cost function CF_P, we sued the following metrics: rl and tl represent the latency for the execution of `read` and `take` operations, respectively;

$$CF_P(p) = w''_1 * rl_p + w''_2 * tl_p + w''_3 * bu_p + w''_4 * mu_p \quad (3)$$

where $\sum w'_i = 1$ with $w'_i \geq 0$, and $\sum w''_i = 1$ with $w''_i \geq 0$.

The weights w_i are used for controlling the contribution of each concern. If the application developer is more interested in availability, she could increase the value of the respective weight to drive the decision of the mechanism towards selecting a combination with more emphasis on the availability policy.

This method minimises resource usage, such as memory and bandwidth, for both availability and performance policies. At the same time, availability and performance are maximised. This method tries to avoid the selection of a sub-optimal combination. If a better combination exists, this method guarantees that it will find it. In fact, the impact of duplicating tuples will be measured by the memory and bandwidth metrics of both CF_A and CF_P.

What makes this method appealing is that the input from the application developer is kept to a minimum. The method autonomously evaluates and selects the best combination requiring from the developer only the choice of the weights. However, finding the optimal combination requires to evaluate all possible combinations of availability and performance policies. The complexity of such method is $O(n * m)$, where n is the number of availability policies and m that of performance.

It is possible to do better with the use of some heuristics. A very simple one could be the following. Availability and performance policies are evaluated separately by the respective cost functions. Each cost function will output the optimal policy for the respective concern, indicated as a_{opt} for availability and p_{opt} for performance. The optimal combination then would be $c(a_{opt}, p_{opt})$. The complexity of this heuristic is $O(n + m)$. We suspect that this heuristic will select optimal combinations as long as the interactions between performance and availability are at a minimum. As a matter of fact, this heuristic selects the optimal policies for each concern separately. Thus, if there are interactions between these concerns this heuristic is not able to take them into account.

A variant of the previous heuristic could be the following. The heuristic uses a concern as a *pivot*. Let us assume that the pivot concern is availability. Using the cost function of availability, the heuristic selects the optimal policy for availability, a_{opt}. Using the general cost function, the heuristic evaluates all combinations in the form of $c(a_{opt}, p)$, where a_{opt} is fixed and p is one of the performance policies. The complexity of this heuristic is $O(n+m)$: $O(n)$ for finding the optimal policy for the pivot concern and $O(m)$ for finding the optimal combination. This heuristic has the same complexity as the previous. However, this heuristic is able to take into account the interaction between concerns. Thus, in some cases it could be more precise than the previous heuristic.

Another heuristic could be the following. The policies of each concern are ranked by using the respective cost functions. After the ranking, the general cost function is used for combining the first x best policies of both concerns. The complexity of this heuristics is $O(n + m + x^2)$: $O(n)$ for ranking the policies for availability; $O(m)$ for ranking the policies for performance; finally $O(x^2)$ for finding the optimal combination.

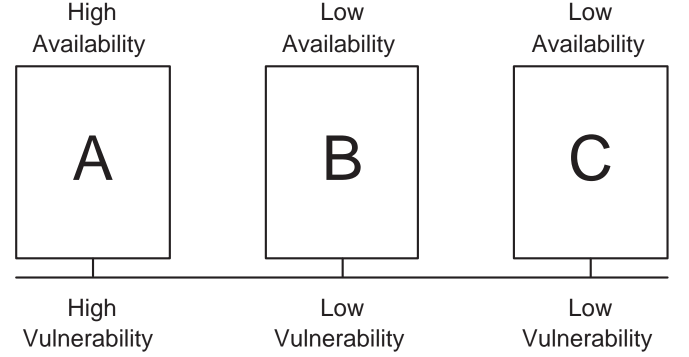

Figure 3: Environment where nodes have conflicting characteristics for different concerns.

These are just a few examples of heuristics. A heuristic offers the advantage of having a lower complexity with respect to a complete evaluation of all combinations. However, this lower complexity could compromise the *precision* of the selection. The higher the precision is, the closer the selection is to the optimal combination. By comparing the selection of a heuristic with that of the complete method, we can quantify the precision of that heuristic. Thus the problem is shifted to finding a heuristic that balances a high precision with a low complexity.

5.2 Availability and Security Interactions

In this section, we analyse the interaction of availability and security concerns. In particular, we are interested in how mechanisms used for availability could interact with security mechanisms. Since performance uses the same type of mechanisms as availability, the following analysis holds also for the interaction of performance and security.

Availability uses a different type of mechanism than security. Mechanisms in availability are distribution strategies that decide *where to place* tuples for providing a given availability. At the same time, the cost of maintaining such availability level should be kept at minimum.

Security is a more pervasive concern. Making a system *secure* (with respect to a threat model) requires taking several precautions scattered across the entire system. For our framework, security mechanisms are concerned with:

- *authenticating* entities that should perform operations on tuples
- *authorizing* the operations on tuples
- *enforcing privacy* of tuples and their content.

The type of interaction between security and availability mechanisms can be explained by the following example. Let us assume that the system is deployed in the environment depicted in Figure 3. In the environment, 3 nodes are provided with different characteristics regarding availability and security. Node A provides a high availability but it is rated as not being secure. On the other hand, nodes B and C together provide a lower availability but higher security. The optimal policy for availability would be to store tuples on node A. However, for security constraints such a node cannot be selected.

This type of conflicts can be handled in two ways: fully automated or with user intervention. In fully automated mode, the system is in charge of taking the decision whether or not an operation that violates security constraints should be executed. For instance, per default no security constraints can be violated. In this case, the system has to resort to another distribution strategy that is sub-optimal but that does not violate security. Although this method is fully autonomous, it could lead to a sub-optimal usage of resources.

When the user is involved in the loop, whenever a conflict arises the user is asked to provide guidance for solving the conflict. The user, based on some knowledge, could temporarily allow the violations of security constraints for the sake of optimal resource usage.

Finally, although availability and security concerns are dealt with by different types of mechanisms, during execution these mechanisms can influence each others. For instance, the encryption of a tuple during the execution of an operation influences the overall latency of the operation. However, this is the price to pay for all policies when security is taken into account.

6. SUMMARY

In this paper, we propose an architecture for a multi-concerns self-managing architecture, where availability, performance and security concerns can be dealt with at the same time. The architecture is such that each concern is isolated from other concerns. We also analysed interactions among the concerns that are currently supported. We differentiated two types of interactions: those that lead to sub-optimal solutions and those that lead to conflicts. For each type, we proposed methods that could be used for dealing with them.

Our previous results showed to certain extend the feasibility of our our approach. To further substantiate our findings, we realise that the work presented in this paper needs more experimental proof. We consider this as the main topic of our future research.

Acknowledgments

This research was supported by the UK's EPSRC research grant EP/C537181/1 and forms part of the CareGrid, a collaborative project with the University of Cambridge. The authors would like to thank the members of the Policy Research Group at Imperial College for their support.

7. REFERENCES

[1] G. Coulson, G. Blair, P. Grace, A. Joolia, K.Lee, and J. Ueyama. "A Component Model For Building Systems Software." In *Proceedings of Software Engineering and Applications*, 2004.

[2] E. W. Dijkstra. "Selected Writings on Computing: A Personal Perspective", pp. 60–66, Springer-Verlag, 1982.

[3] R. Filman, S. Barrettt, D. Lee, and T. Linden. "Inserting Ilities by Controlling Communications." In *Communications of the ACM*, 45(1):116–122, Jan. 2002.

[4] D. Gelernter. "Generative Communication in Linda." *ACM Trans. Prog. Lang. Syst.*, 7(1):80–112, 1985.

[5] D. Garlan, S. Cheng, A. Huang, B. Schmerl, and P. Steenkiste. "Rainbow: Architecture-Based Self-Adaptation with Reusable Infrastructure."In *IEEE Computer*, 37(10):46–54, 2004.

[6] J. Kramer and J. Magee. "Self-Managed Systems: an Architectural Challenge." In *Proc of IEEE Future of Software Engineering 2007 (FOSE'07)*, pp. 259–268, May, 2007.

[7] P. Maes. "Concepts and experiments in computational reflection." In *OOPSLA Š87: Conference proceedings on Object-oriented programming systems, languages and applications*, pp. 147–155, New York, NY, USA, 1987. ACM Press.

[8] P. Oreizy, M. M. Gorlick, R. N. Taylor, D. Heimbigner, G. Johnson, N. Medvidovic, A. Quilici, D. S. Rosenblum, and A. L. Wolf. "An architecture-based approach to self-adaptive software." In *IEEE Intelligent Systems*, 14(3):54–62, 1999.

[9] D. L. Parnas. "On the criteria to be used in decomposing systems into modules." *Communications of the ACM*, 15(12):1053–1058, December 1972.

[10] G. Russello, M. Chaudron, M. van Steen. "Dynamic Adaptation of Data Distribution Policies in a Shared Data Space System." In *Proc. Int'l Symp. On Distributed Objects and Applications (DOA'04)*, volume 3291 of Lecture Notes in Computer Science, Springer-Verlag, pages 1225–1242, Larnaca, Cyprus, October 2004.

[11] G. Russello, M. Chaudron, M. van Steen. "Dynamically Adapting Tuple Replication for High Availability in a Shared Data Space." In *Proc. 7th Int'l Conf. on Coordination Models and Languages (Coordination 2005)*, volume 3454 of Lecture Notes in Computer Science, Springer-Verlag, pages 109–124, Namur, Belgium, April 2005.

[12] G. Russello, M. Chaudron, M. van Steen, W. Stut, and M. Petkovic. "A Personal Health Care System using Secure GSpace." Philips Technical Notes PR-TN 2006/00226, Philips Research Laboratories, Eindhoven, The Netherlands, May 2006.

[13] P. Tarr, H. Ossher, W. Harrison, and S. Sutton. "N degrees of separation: multidimensional separation of concerns." In *Proc. 21st Int'l Conf. on Software Engineering, ACM*, pp. 107–119, NY, 1999.

[14] W. E. Walsh, G. Tesauro, J. O. Kephart, and R. Das. "Utility Functions in Autonomic Computing." In *First International Conference on Autonomic Computing (ICACŠ04)*, 2004.