Engineering Complex Adaptations in Highly Heterogeneous Distributed Systems

Paul Grace, Gordon S. Blair, Carlos Flores Cortes, Nelly Bencomo
Computing Department
Lancaster University
Lancaster, UK
{gracep, gordon, c.florescortes, bencomo}@comp.lancs.ac.uk

ABSTRACT
Distributed systems now encounter extreme heterogeneity in the form of diverse devices, network types etc., and also need to dynamically adapt to changing environmental conditions. Self-adaptive middleware is ideally situated to address these challenges. However, developing such software is a complex task. In this paper, we present the Gridkit self* approach to the engineering of reflective middleware; this embraces state of the art software engineering practices, and flexible dynamic adaptation mechanisms to better support system developers. Domain specific frameworks are modeled and developed to enhance configurability and reconfigurability. We evaluate this approach using case studies in the domains of service discovery and network overlays. These demonstrate the benefits of the approach in terms of aiding and simplifying the process of creating self-configuring and self-adaptive software.

Categories and Subject Descriptors
D.2.11 [**Software Engineering**]: Software Architectures - Patterns (Reflection).

General Terms
Management, Design.

Keywords
Reflection, middleware, adaptation, heterogeneity

1. INTRODUCTION
As software systems become more ubiquitous a new class of large-scale distributed systems has emerged; known as *Systems-of-Systems (SoS)* [1]. These are composed of geographically remote systems that are combined to deliver services that cross device, platform and system administration boundaries. An example of a SoS is a set of wireless sensor networks deployed across rivers to monitor flooding; these send their data to flood prediction models executing on a computational grid in a fixed infrastructure network. SoS of this type are characterised by two fundamental properties:

- *extreme heterogeneity* in terms of interconnected devices (e.g. sensors, mobiles, embedded devices, PCs and clusters) and the communication networks between them (e.g. large-scale fixed networks through to wireless ad-hoc networks);
- *dynamic change*, i.e. the very nature of such pervasive systems means that the operational environment or general context will change over time e.g. due to user mobility, or the fluctuating environmental conditions of wireless networks.

Given these properties, this leads to a degree of complexity that is orders of magnitude greater than traditional distributed systems, and poses new challenges in the field of dynamic and self-adapting distributed systems i.e. what types of adaptive software are required to overcome these problems, and equally how can the developers of such software be better supported. We believe that such challenges are best tackled at the middleware level; and in particular in the form of adaptive middleware [2]. Flexible, configurable and reconfigurable middleware can provide the necessary mechanisms and transparency to allow developers to create SoS that self-configure, self-optimise, self-heal and self-manage.

In this paper we present our experiences of developing Gridkit, which is a self-configuring and self-adapting middleware that can be deployed for applications that face the extreme heterogeneity described previously; the core principles of this work has been previously published [3, 4]; however here we focus on the software engineering methodology we follow to develop self-adaptive middleware. We believe this approach (*the Gridkit self* approach*) and the corresponding software frameworks can be re-used within the field of both adaptive middleware and self-adaptive software in general.

The Gridkit self* approach has three key elements that we will present in detail in this paper:

- *Reflective software frameworks*. These frameworks are composed of components (which may be distributed). Reflective information is maintained about the topology and behaviour of the framework, this allows open introspection and adaptation of the contained elements to support self-configuration and self-adaptation.
- *Flexible adaptation*. Gridkit supports the developer in performing two types of adaptation: i) *node-local adaptation*, where software is adapted on a single node, or ii) *distributed adaptation*, where the software to be adapted may be spread across multiple hosts. Given the nature of SoS,

AUTONOMICS 2008, September 23-25, Turin, Italy

DOI 10.4108/ICST.AUTONOMICS2008.4647

these will likely be performed in different conditions (e.g. in a local area network versus in a wireless ad-hoc network). Therefore, a one size fits all mechanism for adaptation is infeasible; hence we support the use of flexible adaptation mechanisms.

- *Domain Specific Engineering* is the practice of collecting past experience in a domain in the form of reusable assets. Hence, domain experts design and implement software components and software patterns to better support configuration and adaptation in the face of heterogeneity and dynamic change within their particular domain.

We evaluate the Gridkit self* approach by considering two distinct domains of middleware behaviour. We use case study based evaluation for both service discovery middleware and network overlays; these illustrate how different software artifacts (in terms of architectural patterns and transition models) are created per domain; and demonstrate the benefits of this philosophy in optimizing the configuration and adaptation of software, and also qualitatively in terms of simplifying the task of systems developers.

The remainder of the paper is structured as follows. Section 2 introduces scenarios from the field of SoS and documents and challenges for research in this field. Section 3 introduces the Gridkit self* approach. Section 4 presents the case-study based evaluation. Section 5 discusses related work, and finally concluding remarks and future work are in section 6.

2. THE CHALLENGES OF SOS

This section analyses two pervasive Systems-of-Systems scenarios that are characterized by the need for multiple middleware services to be deployed across a integrated, heterogeneous network types.

2.1 Command & Control Systems

This scenario is focused on forest or savannah fire fighting in remote, poorly resourced locations. There are two user roles involved: *controllers* and *fire fighters.* Controllers manage the operation: they move fire fighters, issue commands, decide where to deploy fire sensors, and investigate real-time simulations that predict the spread of the fire. Fire fighters use mobile devices, on which graphics-based commands from controllers are displayed, and deploy wind speed sensor networks.

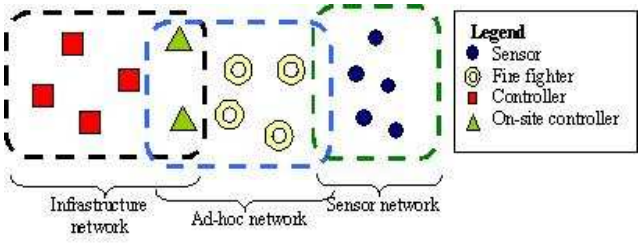


Figure 1. Extreme heterogeneity in the fire fighting scenario.

Figure 1 illustrates a typical network configuration for this scenario: fire fighters form ad-hoc connections between themselves, sensors and on-site controllers; and infrastructure networks connect all controllers. Depending on the location of the fire, the connections between on-site and remote controllers could be provided by satellite, GPRS, Wireless LAN, or other network types.

In terms of middleware, this scenario requires a group communication service to enable controllers to disseminate commands to fire fighters (where to move, which part of fire to fight, where to put sensors, etc.); and a publish-subscribe service is required for the collection of sensor events to be fed to the controllers' fire spread simulations.

2.2 Environmental Monitoring

In this scenario, a river, estuary and bay are instrumented with sensors to monitor temperature, water level, flow rate, pollution levels etc. Some of these sensors are networked using Ethernet (e.g. sensors in tidal-defence walls), while others employ wireless technologies (e.g. IEEE 802.15.4 or 802.11 radios; or long-wave radios for underwater use). These may be mobile in the water, and will come into the range of fixed sensors in an ad-hoc fashion. Point-to-point microwave connectivity may also be used to link individual sensor networks to strategically placed IP gateways at which sensor data is collated and cached.

Given this infrastructure, scientists in widely-dispersed locations selectively store data for future analysis; integrate and process live sensor data on their workstations; cooperatively visualise this data in real-time (supported by a video conferencing system); and use both stored and live data to computationally steer long running environmental simulations. Figure 2 illustrates a typical network configuration for this scenario. The middleware requirements for this scenario are: i) a data retrieval service is needed to collect data from the sensor networks, ii) an event service is needed to 'push' sensor data to long-running simulations, and iii) a streaming service is needed to support video conferencing between fixed workstations.

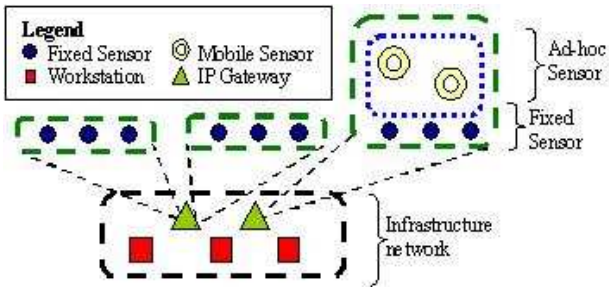


Figure 2. Extreme heterogeneity in the environmental informatics scenario.

2.3 Analysis

From these scenarios it can be seen that there are many challenges to resolve. Firstly, how can the functional middleware services (e.g. resource discovery, publish-subscribe and others) be developed and deployed across diverse environment conditions. Secondly, how can adaptations be performed to respond to changing conditions. Thirdly, how can the developers of this software be better supported in the face of complexity. Finally, how can non-functional concerns such as security and dependability be achieved in an end-to-end manner. In this paper, we aim to provide initial answers to the first three of these.

3. THE GRIDKIT SELF* APPROACH

3.1 The Gridkit Middleware

Gridkit [4] is a novel reflective middleware framework that deploys an extensive and extensible set of middleware services over an infrastructure of overlay networks which it itself creates and manages. Gridkit is designed with a philosophy that promotes openness as the underlying principle in engineering self* behaviour; this combines three core technologies: *reflection*, *software components* and *component frameworks* [2].

Software components act as the building blocks of middleware, these third party deployable elements [5] promote configurability, re-configurability and re-use at the middleware level. Reflection is then used to provide a principled mechanism to inspect the current system behaviour in order to inform decisions that dynamically adapt the component structure at run-time. Finally, component frameworks (discussed in greater detail in section 3.2) constrain the design space and the scope for evolution within particular domains of middleware behaviour; a component framework is generally defined as a collection of rules and contracts that govern the interaction of a set of components [5]; in our case it further acts as the managing entity for self* behaviour.

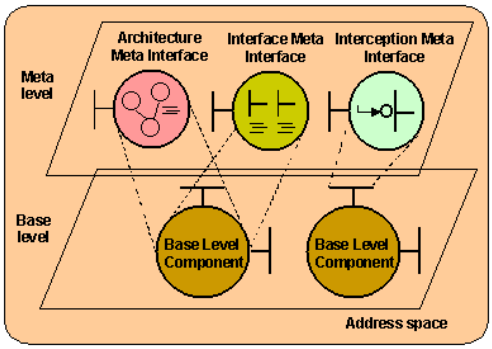


Figure 3. The meta-space structure of OpenCOMJ

Gridkit is built in terms of a Java-based reflective component model called OpenCOMJ (http://gridkit.sourceforge.net). This presents a runtime kernel that supports the loading and binding of lightweight software components at run-time. OpenCOMJ also supports basic reflective operations (these can be extended, as described in section 3.2); figure 3 illustrates the three core meta-space behaviors: *interface*, *architecture*, *and interception*. The interface and architecture meta-models provide structural reflection in terms of inspecting the interfaces of components, and the topology of components in terms of connected elements; the interception meta-model supports behavioural reflection by enabling the dynamic insertion of interceptors, which support the insertion of pre- and post- behaviour on to interfaces.

The overall software architecture of Gridkit is illustrated in figure 4. Atop the component model is a set of middleware frameworks that can be composed to overcome heterogeneity across diverse conditions. The *overlays framework*, which is a distributed framework for the deployment of multiple overlay networks. In practice, this amounts to hosting, in a set of distributed overlay framework instances, a set of per-overlay plug-in components. This framework provides a virtualized view of network behaviour (in potentially many different environments) to allow higher level services and frameworks to be easily deployed without being concerned about the underlying network heterogeneity; section 4.2 describes the development of this framework in more detail.

Above the overlays framework is a set of further "vertical" frameworks that provide functionality in various orthogonal areas, and can optionally be included or not included on different devices. In brief, the frameworks are as follows: the *interaction framework* accepts multiple interaction type plug-ins (e.g. RPC, publish-subscribe, group communication); the *service discovery framework* accepts plug-in strategies to discover application services (e.g. SLP, UPnP, Salutation); the *resource discovery* framework accepts plug-in strategies to discover resources such as CPUs and storage (e.g. peer-to-peer search); the *resource management* and *resource monitoring* frameworks are respectively responsible for managing and monitoring resources; and the *security* framework provides general security services for the rest of the frameworks.

These illustrate a central philosophy of capturing domain behaviour within individual frameworks in order for them to be developed and optimized individually by domain experts. Hence, the internal architectures are likely to be significantly different; however, the component-philosophy of Gridkit allows these frameworks to be composed to fit a wide range of requirements. This is illustrated in the case-studies of section 4, where we focus on the development of service discovery behaviour.

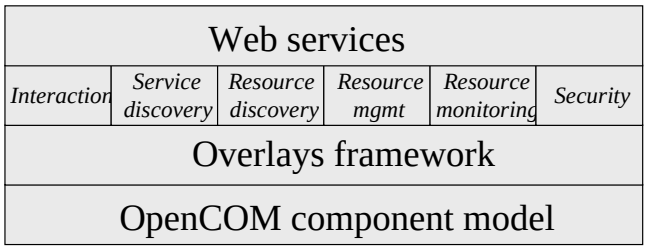


Figure 4. The Gridkit architecture

3.2 Flexible Adaptation

As previously described, Gridkit provides software frameworks as the tools to perform and manage adaption. However, self* systems will typically require a wide range of adaptation types e.g. adapting software within an address space compared to adapting software across machines in a coordinated manner; and these will take place in diverse operational conditions e.g. across PCs in a fixed network or on sensors in a wireless ad-hoc networks. Hence, we believe that a one-size fits all method is infeasible, and Gridkit therefore provides developers with flexible, extensible frameworks that can be tailored to the requirements. Here, we present example mechanisms for performing node-local and distributed adaptation.

3.2.1 Node-local Adaptation

The local software framework model (illustrated in figure 5) is based upon the concept of composite components. Each framework is an OpenCOM component that has internal architecture. Additionally, each framework supports the following dimensions for performing safe, valid reconfigurations in the local address space: i) an architecture meta-protocol, ii) validated

reconfigurations, iii) quiescence management, and iv) policy configurators.

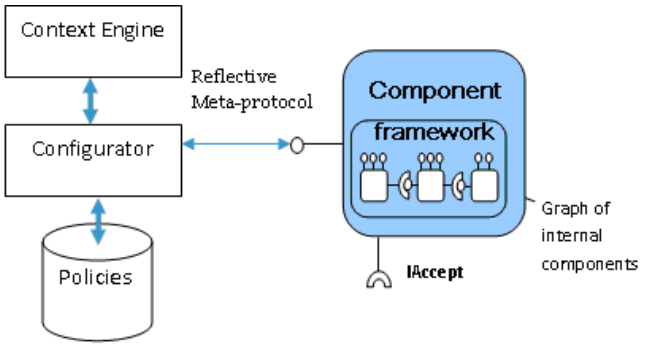


Figure 5. Frameworks for Node-local Adaptation

The *architecture meta-protocol* supports introspection and dynamic reconfiguration. Each framework maintains a local 'graph' representing the internal structure. The protocol operations act on and manipulate this meta-graph; the components and their connections can be viewed, components can be added, removed, etc. Any changes to the graph are then reflected in the concrete components.

Validation of reconfigurations. Providing open access to the structure of a system, and the ability to make run-time changes, increases the likelihood of system failure and opens it to third party attack. To guard against this, each framework exports a 'health check' mechanism (illustrated in figure 5 as the required interface called *IAccept*); components encapsulating knowledge about valid dynamic reconfigurations for this particular framework are then plugged into this interface. Each reconfiguration is applied as a local transaction; hence once committed, a reconfiguration is validated such that invalid attempts are rolled back to the previous safe state.

Quiescence Management. Reconfiguration operations must only be carried out when a framework is in a safe quiescent state. If a change to the configuration is made while one or more service calls are executing, then the results of these invocations could be compromised or lost. Therefore, each framework provides a readers/writers lock for access to the local graph. Each service call accesses the lock as a reader (there can be *n* readers using the lock at any time). Any reflective call accesses the lock as a writer (a single writer can access the lock when there are no readers). Interceptors are used to manage access; attached pre- and post-methods implement the roles of a readers/writers solution. E.g. a pre method accesses the lock and increments the reader count, while the post method decrements the count and if it is the last reader the lock is released for writers.

Configuration Management. Local frameworks use the configurator pattern [6] to perform adaptations. A configurator acts as a unit of autonomy for making decisions about when and how to change the framework. This maintains a set of local policies, in the form of configuration patterns and adaptation rules; these are described in XML and use the Event-Condition-Action style. When an event is detected (typically a context change from the context engine), it applies the action i.e. configure the framework, or perform a transition from one topology to another. Such actions are enacted by invoking operations on the architecture meta-protocol.

3.2.2 Distributed Adaptation

Adapting middleware may require software to be adapted across hosts in a co-ordinated manner. For example, to change the behaviour of a multicast protocol, the software at each participating node must be reconfigured. Here we discuss how Gridkit supports distributed adaptation using *distributed software frameworks*. The model for local reconfiguration is equally applicable to distributed component topologies; the central themes of architecture meta-protocols, validation, quiescence and policy-driven configurators are followed. This allows the domain engineering principles described in the next section to be applied equally across both cases.

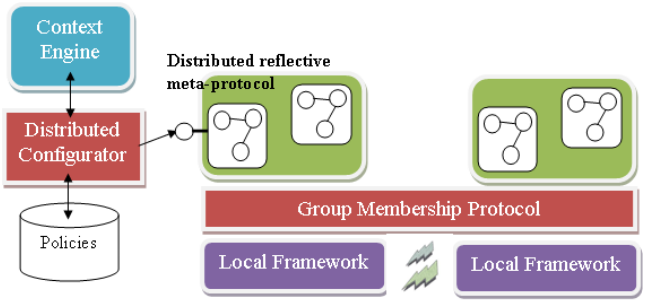


Figure 6. Distributed Frameworks

Reflective meta-protocol. Each distributed framework maintains reflective information about the node members and the component topologies on these nodes. A lightweight group membership service serves as the base mechanism for distributing meta-data (illustrated in figure 6); this data then builds the view of the system wide architecture. The group protocol is customizable: typically different group membership overlays will suit different domains (e.g. one for internet scale, versus one for ad-hoc networks). The distributed reflective meta-protocol is available from each instance of the framework (seen in figure 6); the set of available meta-operations allows manipulation of the distributed graph to enact adaptations. Note, the distributed framework, changes the local instance through the local reflective meta-protocol. Hence, it is a hierarchical architecture.

Validation of a distributed framework is important to ensure that the collaborating nodes maintain a correct implementation of the middleware across nodes. Hence, in a similar manner to local validation, after a distributed adaptation has taken place this update is checked through inspection of the meta-data. Designated nodes in the framework have a set of plug-in rules that are used to validate the integrity of component updates across multiple nodes. An invalid reconfiguration can thereby be detected and repaired.

Centralised Quiescence. For safe dynamic reconfiguration it is important to ensure that updates do not impact the integrity of the system. Hence, the distributed framework must be made safe to adapt, i.e. placing it in a quiescent state. We have so far developed a single, centralised implementation for deriving a safe state in the distributed framework that is based upon the local host approach. A request to reconfigure the distributed framework from a central node generates a request message asking each local node to be placed in a quiescent state; this message is propagated

via gossiping through underlying group service. Once a local framework is in a quiescent state it returns a notification to the configurator node. Upon the condition that all members are in a quiescent state the reconfiguration can take place. The disadvantage of the centralised approach is that it may be too resource intensive, and may not scale suitably for large numbers of nodes. Additionally, it may not be necessary to place all nodes in a safe-state at the same time, or have a single node managing the transition to a safe state. Hence, the frameworks should support *selectable approaches* to safe-state management that can be tailored to the particular style of reconfiguration to be performed in the environment that the framework is deployed.

Policy-based Configurators. Distributed configurators (as seen in figure 6) again follow the same pattern as in local frameworks. They receive events about changing environmental conditions from a context engine, select policies, and then perform distributed reconfigurations. However, distributed frameworks may have more than one configurator (e.g. there could be one on every node). Therefore, consensus protocols must be used to ensure that all members of the framework agree on the action to perform. Our development of the reconfiguration mechanisms has so far concentrated on centralised configurators; however, we are also investigating the introduction of selectable and replaceable consensus algorithms.

3.3 Domain Engineering

We have presented Gridkit's capabilities to perform adaptation. However, we acknowledge that the development of self* systems that make use of these facilities is becoming increasingly complex. Developers must deal with a large number of variability decisions when planning the configurations and dynamic reconfigurations. These include decisions such as what components are required, and how these components must be configured and changed according to variations in the environment and context. Hence, in this section we discuss a domain engineering methodology that we have successfully followed in developing different Gridkit frameworks.

The overall methodology is illustrated in the workflow diagram of figure 7. The key contributor and initiator is the domain engineer; this may be one or more people who have expert knowledge in a particular field of middleware behaviour, and it is their task to create a re-usable software framework that overcomes the problems of extreme heterogeneity and dynamic change in that particular domain. For this they must produce a set of artifacts that can be introduced into Gridkit.

First, the domain engineers use a range of modeling tools known as Genie; further information about the features and software engineering benefits of these tools is available from [7]. Two key software artifacts are produced:

i) *Software architecture Patterns.* These describe generalized configurations of components that are suitable for different environment conditions i.e. if the framework encounters particular heterogeneity conditions then component configuration A conforming to the pattern is employed, in a different case configuration B is chosen.

ii) *Transition Models.* These describe the changes that must be made to a component topology when a change in environmental context is encountered. They are typically related to the initial configuration patterns (to avoid model duplication). They also capture the condition (context or requirement change) that requires the reconfiguration to be enacted.

Once the framework has been modeled the tools generate the corresponding XML policy files i.e. the software architecture patterns become configuration policies and the transition models become reconfiguration rules. These are then deployed into a Gridkit framework without further implementation (note this applies equally to local or distributed software frameworks).

The domain engineers also directly design and implement the software components that comprise the domain specific middleware behaviour; this will typically be performed in parallel with the modeling. Indeed we do not see this as a traditional waterfall process; typically the artifacts will be refined as new requirements are captured, and further experience of the domain is discovered (e.g. once the system has been deployed). Once complete, the components can be directly deployed to a Gridkit framework.

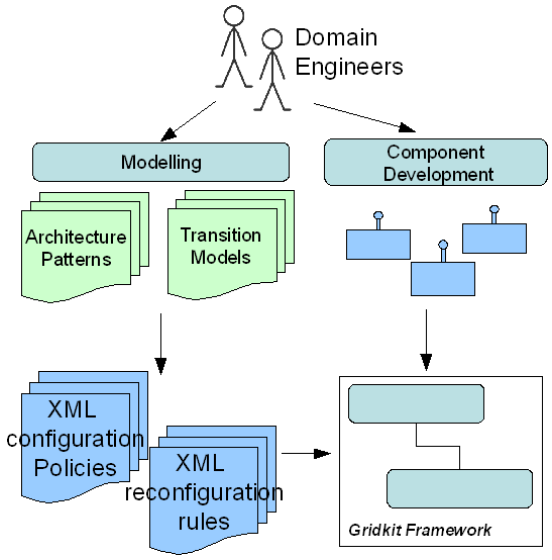


Figure 7. The Gridkit Self* Engineering Methodology

It can be seen from this process that the domain engineers are shielded from many of the complexities of developing self* behaviour. They do not need to code adaptation behaviour, nor do they need to write reconfiguration policies. Instead, the use of models as first class entities raises the level of abstraction. Illustrations and examples of the modeling tools will be described in the service discovery and network overlay case studies in the next section.

4. EVALUATION

4.1 Service Discovery Case Study

In pervasive applications there is no prior knowledge of what resources are available in the environment or what method should be used to communicate with them, hence discovering the appropriate services in these environments is challenging. Many

service discovery protocols have emerged to solve this problem, with heterogeneous solutions in each environment type; for example, discovery protocols for fixed infrastructure networks e.g. SLP, UPnP and Jini; and discovery protocols for ad-hoc networks e.g. ALLIA [8], GSD [9] and SSD [10]. The goal of this case study is to illustrate how the Gridkit self* approach supports the developer in creating a discovery framework that can operate in heterogeneous environments, and dynamically adapt it behaviour to changing conditions.

To advertise and discover services, a discovery platform utilizes: i) a User Agent (UA) to discover services on behalf of clients, ii) a Service Agent (SA) to represent and advertise services, and iii) a Directory Agent (DA) to support a service directory where SAs register their local services and UAs send their service request. Hence, a DA is capable of storing temporal service advertisements, matching requested services against advertisements stored in the cache and replying to requesting clients when a positive match is found.

4.1.1 *Software Architecture Patterns*

Figure 8 illustrates the core component pattern designed by a domain expert to be used in development of all discovery protocols within the framework. The *advertiser* component: i) broadcasts advertisement messages, ii) maintains a service directory overlay (dependent on the protocol, and iii) manages cached data. The *request* component constructs and sends request messages. The *reply* component constructs and sends response messages. The *network* component handles routing of messages (this can be replaced by the Gridkit overlay framework). The *policy* component enforces user preferences, application needs and/or inclusive context requirements. Finally, the *cache* component stores messages and advertisements for later use by the protocol. More details about the requirements and behaviour capture of this pattern are described in [11].

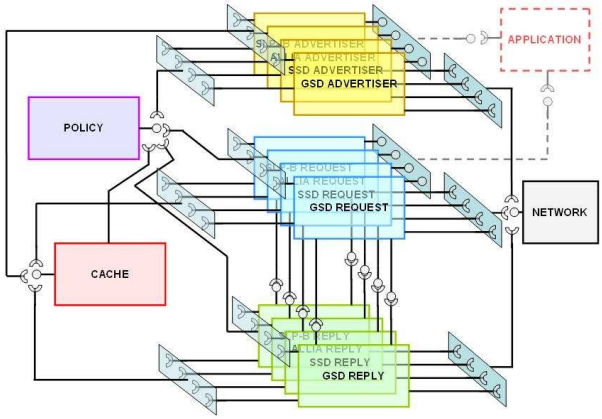
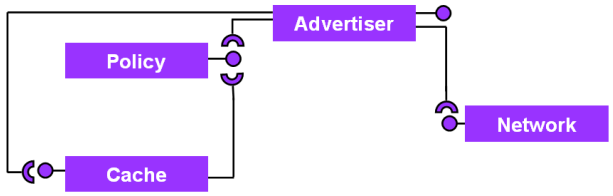


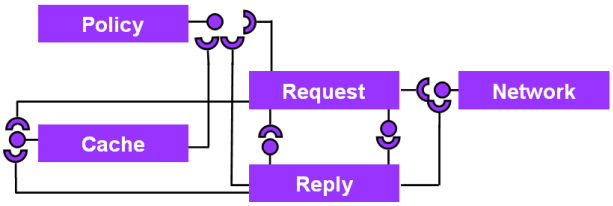
Figure 8. The Service Discovery Framework Pattern

Moreover, further architecture patterns were designed to support individual discovery agent personalities on any implemented discovery platform. Figure 9 demonstrates how the framework can be configured to support either a SA or UA personality by restricting the number of components to only those required to provide a determined functionality.

Domain experts then developed the components (to match the architecture pattern) for four discovery protocols: SLP, ALLIA, GSD and SSD. Three of the component types are common to all protocols, hence, only three (advertiser, request and reply) need to be implemented for each protocol. This is illustrated in figure 8; the four protocols are configured side-by-side and share the common components. Hence, the pattern promotes *re-usability* across configurations and reduces the implementation task.



a) SA framework configuration



b) UA framework configuration

Figure 9. Service Discovery Agent Patterns

4.1.2 *Transition Models*

Within the domain of service discovery there are many opportunities to perform self adaptation (in particular these are *node-local adaptations*, performed only on an individual participant in the discovery protocol):

- Heterogeneity change e.g. a device moving from a fixed network to a wireless ad-hoc network requires one protocol to be replaced by another e.g. SLP replaced by ALLIA.
- Role change e.g. if the system size increases (for scalability reasons) reconfigure the SA configuration to DA.
- the use of a different role strategy to save energy e.g. If a node DA has low battery and it was originally a node with the role SA, the node should be reconfigured to its original SA configuration to reduce its processing.

To design and implement such reconfigurations, the domain engineer creates transition models that describe how one of the patterns from 4.1.1 is transformed to another pattern (e.g. UA to DA). To do this, an adaptation is defined as the process of having the system going from a given configuration C_i to another configuration C_j given the conditions of the context T_k. This is modelled using transition diagrams. A screenshot of the Genie tool that is used to specify these is shown in figure 10. An adaptation policy is associated with the relationship (arc) between the configuration for the variant UA (Ci) and the configuration for DA (Cj) for a given context T_k specified by the policy. Hence, for each arc an XML reconfiguration rule is generated that is understood by the Gridkit framework.

An example reconfiguration rule (described in pseudocode rather than XML to conserve space) is:

```
if ( RSA ) then
        reconfigure(UA,SA)
end
```

Based upon the requirement for the protocol to now advertise services (RSA context) and the current configuration is UA, it transforms from UA to SA. The framework will take the information from the rule and use a series of reflective operations to determine what the configuration is (protocol type, and to verify it is UA) and then performs node-local adaptation via meta-operations to create the SA configuration.

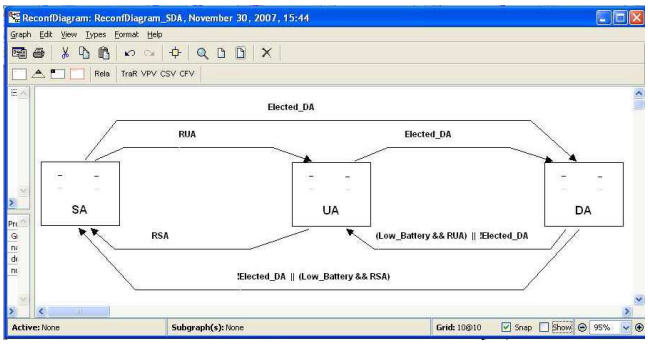


Figure 10. Transition Model for Service Discovery in the Genie Tool

4.1.3 Analysis

The original hypothesis of this case study was that the Gridkit self* approach supported the developer in overcoming the challenges of heterogeneity and the need for dynamic adaptation. Here, we discuss the two dimensions of self-configuration and self-adaptation. We also investigate the costs of this approach.

Configurability. Heterogeneous discovery platforms are implemented to a common pattern. This simplifies the configuration process since the component types and connection bindings remain the same for any protocol implementation. Hence, based upon the operating conditions, it is straightforward to configure the appropriate protocol; this shields developers and users from the complexities of heterogeneity. Further, configuring minimal agent personalities brings benefits; for instance, when a discovery protocol with a structured distributed directory architecture is utilized many nodes will only need to discover services, and a UA configuration can be employed. Hence, by configuring individual protocols according to the role (UA, SA or DA), the resources required can be reduced.

Reconfigurability. The modeling of reconfigurations hides many of the complexities of developing adaptive software. Indeed these can be designed and employed at runtime after the protocols have been deployed; this is because Gridkit frameworks provide a strong separation between the implementation of the protocol and the mechanisms for adaptation. To tackle context change in heterogeneous environments, fine-grained and coarse-grained changes can be made. The complete protocol can be changed if it no longer functions in the environment; and finer-grained role changes can to respond to environmental context or application requirement changes.

Resource Overhead. To analyze the overhead of our framework we measured the size of the Java classes (that made up the component configurations) as loaded into memory. These measures are illustrated in table 1; these figures show the cost of each individual protocol in the framework. Then we measured the cost when multiple protocols are configured. We compared these measures against the side-by-side measurement of individual protocols (not configured in the framework). It can be seen that resource usage is reduced (due to component re-use from the pattern properties), and that the overhead of a multiple protocol personality is not restrictive for resource-poor devices. Hence, tackling heterogeneity does not come with prohibitive costs.

Table 1. Memory overheads of discovery framework

				Framework	Side by side
ALLIA	SSD	GSD	SLP-B	SIZE(KB)	SIZE(KB)
X				66.96	
	X			70.46	
		X		64.26	
			X	89.26	
X	X			129.16	137.42
X	X	X		172.56	201.68
X	X	X	X	209.76	290.94

4.2 Network Overlays Case Study

As well as needing to run effectively over an ever-increasing range of networking technologies (e.g. large-scale fixed networks, mobile ad-hoc networks, resource impoverished sensor networks, satellite links, etc), distributed applications are increasingly demanding sophisticated and application-tailored services from the underlying network (e.g. multimedia content distribution, reliable multicast, etc.) *Network overlays* provide an approach to the virtualisation of the underlying network resource(s), making it possible to provide a range of different networking abstractions including peer-to-peer groups, distributed hash tables, application-level multicast, etc.

In this case study we describe the development of the *open overlays* framework. The goal is to demonstrate the benefits provided by Gridkit frameworks in creating a configurable and reconfigurable framework that supports (flexible) *virtualization* of the network resource, the *co-existence* of multiple (physical or) virtual networking abstractions, and potentially support the *layering* of virtual network abstractions to address the challenges of heterogeneous network environments.

4.2.1 Software Architecture Patterns

Figure 11 illustrates the general patterns for overlays defined by domain experts. This is a two-level architecture:

i) *Overlay plug-ins* are per-node implementations of network overlays. For example, Figure 11 shows four overlay plug-ins: TBCP, Scribe, and plug-ins for a Chord Distributed Hash Table (DHT) and a Chord Key-Based Routing (KBR) overlay. Multiple overlays can operate simultaneously in the framework either in mutual isolation (cf. TBCP and Scribe) or in a stacking relationship (e.g. Scribe and Chord DHT are both stacked atop Chord KBR).

ii) The *overlay pattern***.** Overlay plug-ins are themselves 'mini' frameworks composed of three distinct elements that respectively encapsulate the following areas of behaviour: i) *control* behaviour, in which the node co-operates with its

peer control element on other nodes to build and maintain an overlay-specific virtual network topology; ii) *forwarding* behaviour that determines how the overlay will route messages over the aforementioned virtual topology; iii) *state* information that is maintained for the overlay; e.g. nearest neighbours.

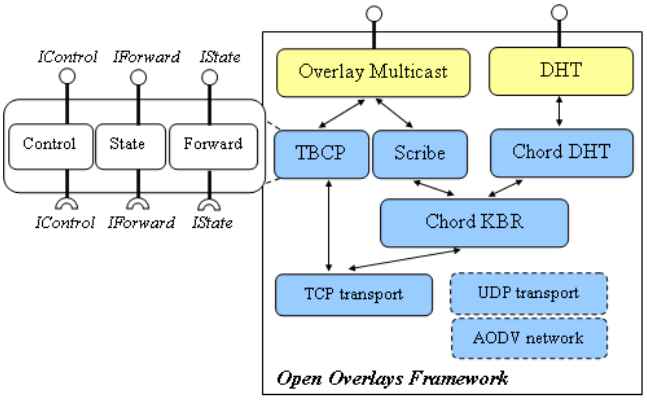


Figure 11. An example configuration of the open overlays framework

Hence, the framework can be instantiated with many possible configurations to meet wide variation in heterogeneous conditions e.g. if multicast is required in an ad-hoc network, then an appropriate overlay is selected (e.g. gossip-based).

4.2.2 Transition Models

The overlay pattern was designed by the domain engineers to facilitate the complex process of adapting network overlay behaviour e.g. the state component maintains the distributed state model (and hence this does not need to be transferred to new components). The topology of the network can then be changed by performing a *distributed adaptation* of control components on each host, or similarly the routing behaviour component can be altered by adapting the forwarder component on each host. Hence, the transition models are straightforward manipulations of the overlay pattern; figure 12 illustrates one example of a control component change. Here, a multicast overlay for resource poor nodes in a fixed wireless network is adapted; the SP box represents a shortest path spanning tree overlay plug-in; FH represents a fewest hop spanning tree configuration. They differ only in their control component.

A fewest hop tree is more resilient to node failure; hence, if the failure rates of nodes in the network increases then the shortest path tree (whose behaviour is severely affected by failure) topology is changed to a fewest hop tree topology. However, a shortest path tree consumes less power, hence when there is minimal failure and power is low then the network reverses. In the model, the designer states the transition, context event and also the style of adaptation; in this case the centralized co-ordination framework is selected to place the network in a safe state and ensure that the adaptation is coordinated. The model is used to generate XML policies that inform distributed adaptation.

4.2.3 Analysis

Ease of use. The overlay patterns have been used by 15 programmers, from a range of institutions, with different levels of programming experience, in a number of system development projects (e.g., projects developing middleware for sensor networks, resource discovery, and publish-subscribe) to develop overlays. From observation and discussion we were able to draw conclusions about the ease with which the approach helped develop adaptive software. Plug-in developers generally understood and followed the approach implied by the overlay pattern, and to this extent their solutions were easily deployable, configurable and adaptable. A typical overlay plug-in is developed in a time frame of 2 to 8 weeks depending on the complexity of the overlay. Framework users found it relatively easy to apply the existing profiles of the framework. Hence, despite the fact that the evidence is primarily anecdotal, and that there are areas of possible improvement, we believe that it is reasonably safe to conclude that third parties can follow the approach with relative ease.

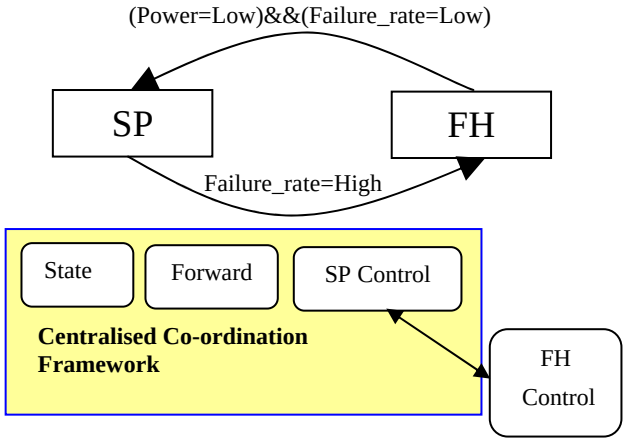


Figure 12. Topology Adaptation of an Overlay Network

Configurability. To measure the extent of the configurability of the framework we calculated the numbers of possible configurations in each of three profiles (i.e. an 'empty' profile consisting of only the framework itself, a 'multicast' profile for multicast overlays, and a 'full' profile containing all of the networks we have developed so far). The numbers, which are summarised in the rightmost column of Table 2, result from an exhaustive enumeration of all the configurations. The results show that the more complex and well-populated profiles support a very large number of possible configurations; e.g. the 'full' profile has 26,999; this does not mean that programmers must write 27,000 rules, rather the approximately 30 patterns for the full profile combine to offer many potential configurations; that is the framework self-configures horizontal and vertically to create these combinations.

Table 2. Configurability and overhead results

Profile	No. plug-ins	No. config. rules	Disk mem. for config. rules (KB)	Disk mem for plug-ins (KB)	Total No. of configs available
Empty	0	0	0	60	1
Multicast	21	19	59	169	89
Full	40	31	87	252	26,999

Furthermore, the overlay pattern contributes significantly to the configurability of the framework by supporting fine-grained configuration of individual overlays. Consider, for example, a Gnutella implementation with either a random-walk-based, or a flooding-based forwarder; or a tree overlay with a control element that either contains or doesn't contain a self-repair algorithm. This applies equally when the overlay pattern is decomposed.

Resource overhead To assess the price paid for the use of software architecture, we quantified the resource overhead incurred by the open overlays framework in three experiments. All of these employed components from Gridkit 1.5/ OpenCOM v1.3.5 (available from http://gridkit.sourceforge.net), executing on a Java 1.5.0.10 virtual machine on a networked workstation with a 3.0 GHz Pentium 4 processor, 1 Gbyte of RAM and running Windows XP. The experiment (see Table 2) investigated the *static storage footprint* costs of each profile; i.e. the disk space required to store the framework, components and configuration rules. This measure is important as it illustrates the cost of storing not only a starting configuration but also any reconfigurations that may subsequently be applied. It can be seen from Table 2 that the base framework requires 60K before any plug-ins are added. Note that the configuration rules take a lot of storage (usually at least 2KBytes) because they are coded in XML.

5. RELATED WORK

There are a number of related areas of research to this work. These consist of software engineering approaches to self* software, reflective component models, reflective middleware, and alternative approaches to distributed adaptation of network protocols and middleware. We now analyse these in turn, examining how they differ from our approach.

Reflective middlewares and component models, e.g. Fractal [12], OpenORB [13] and DynamicTAO [14] came to prominence in the past decade. Generally these provide the infrastructure to adapt; typically node-local adaptation according to a local policy. Although potentially suitable for supporting some classes of self-managed systems, the dimensions of co-ordinated, distributed adaptations have not been addressed; therefore, we believe Gridkit promotes improved support for a wider range of systems e.g. SoS and decentralized classes of self-managed systems.

One alternative component approach that has investigated the coordinated reconfiguration of decentralized, self-managed systems is k-Components [15]. Here, a k-Component is a component with local architecture and a reflective meta protocol to inspect and adapt this architecture. Each k-Component is then related to a management agent; this is responsible for monitoring the environment and making decisions about when to adapt the component structure. In the co-ordination dimension, distributed agents can communicate with one another, although decisions to adapt are made locally. Hence, the approach is suited to only decentralized reconfigurations, with no guarantee that behaviour is changed across a system. Our approach, is in general more flexible allowing the mechanism for co-ordinated adaptation to be tailored to the requirements e.g. centralized or decentralized.

NecoMan [16] offers an alternative approach to dynamic reconfiguration, whose capabilities have inspired many of the features of our approach. It supports safe, co-ordinated updates of distributed services, typically related to network protocols. However, it has not yet been applied in diverse application environments to illustrate its full flexibility; however, we believe it presents many interesting mechanisms that could be applied within our frameworks; especially our points of flexibility in terms of consensus and quiescence. The NecoMan approach has been extended to also manage aspect compositions (rather than components); DyRES [17] provides flexible algorithms to ensure that aspects are safely adapted in a co-ordinated manner.

Silva et al. [18] present a framework to support the automatic self-adaptation of distributed application components. Our approach follows some of their key ideas: monitoring the current system state, supporting flexible algorithms for diverse conditions, and using the configurator pattern. Our approach differs by targeting frameworks of self-managing middleware elements, as opposed to application components. In addition, we consider an architectural view of distributed frameworks, with principled reflection mechanisms to further support adaptation decisions. Hence, self-adaptation can be applied on demand at different levels of the distributed system, from the network protocols, to the communication middleware, to the applications.

A common theme of related work in self* systems is that they provide the mechanisms to perform adaptation; however, they do not provide the developer with additional software engineering methodologies to deal with the inevitable complexities that follow. This is now identified as a key problem; indeed three seminal papers [19][20][21] call for new engineering approaches in this field and from the middleware community. We believe that the work in this paper provides initial solutions to overcome a subset of the identified issues, especially in providing solutions to complex adaptations.

In terms of model-based approaches, MUSIC [22] presents a modelling framework for the specification and management of dynamically adaptive systems. Essentially, this offers a different configurator pattern where adaptations are made to ensure a utility function is achieved. Our approach is complementary, and it would be fruitful to investigate the use of this pattern within our flexible frameworks. MADAM [23] presents a modelling approach conceptually similar to our approach; whereby component configurations and transitions are modelled. However, by capturing domain expertise and adding it to models, we believe systems are in a better position to handle the problems of SoS.

6. CONCLUSIONS AND FUTURE WORK

In this paper we have presented the challenges that are faced by the developers of self* software: extreme heterogeneity and dynamic change. We described the Gridkit self* approach; this is composed of the Gridkit middleware framework which offers the developers a flexible set of software tools for performing node-local and distributed adaptation, underpinned by principled reflective mechanisms. Combined with this, we suggest a domain engineering approach to design and build individual middleware functionality; the use of architecture and transition models provides suitable abstractions to supports developers create configurable and reconfigurable systems. Case studies illustrate the strong benefits of utilizing software patterns and domain expertise to implement self-adapting middleware. It can be seen that the developed service discovery and overlay frameworks are highly configurable and can therefore respond to the challenges of heterogeneity; the ability to adapt is considered throughout the development lifecycle. Using domain expertise, complex adaptations can be rapidly developed and in many cases re-used.

We see future work in two key areas. Firstly, we do not currently consider verification and validation of configurations and adaptations. Here we envisage the use of modeling tools and simulators that allow developers to test and pre-validate reconfigurations before they are deployed. Secondly, it is likely that in complex systems emergent behaviour will be observed; hence we plan to investigate this behaviour further and extend our engineering philosophy to cope with such problems.

7. ACKNOWLEDGMENTS

Our thanks to the ESF and Minema for funding contributions to this work. Gridkit has involved many; therefore the authors would also like to thank: Geoff Coulson, Francois Taiani, Barry Porter, Danny Hughes, Phil Greenwood, Chris Cooper, David Duce, Wei Cai, Musbah Sagar, Jason Li and Gareth Tyson for their input.

8. REFERENCES

[1] A. P. Sage, C. D. Cuppan, “On the Systems Engineering and Management of Systems of Systems and Federations of Systems”, Information, Knowledge, Systems Management 2(4): 325-345, 2001.

[2] G. Blair, G. Coulson, L. Blair, H. Duran-Limon, P. Grace, R. Moreira, N. Parlavantzas. “Reflection, Self-Awareness and Self-Healing”, In Proceedings of Workshop on Self-Healing Systems '02, Charleston, SC, November 2002.

[3] P. Grace, D. Hughes, B. Porter, G. Blair, G. Coulson, F. Taiani, “Experiences with Open Overlays: A Middleware Approach to Network Heterogeneity”, In Proceedings of the 3rd ACM International EuroSys Conference '08, Glasgow, Scotland, April 2008.

[4] P. Grace, G. Coulson, G. Blair, B. Porter, “Deep Middleware for the Divergent Grid”, Proceedings of the 6th IFIP/ACM/USENIX International Middleware Conference 2005, Grenoble, France, November 2005.

[5] C. Szyperski, “Component Software, Beyond Object-Oriented Programming”, ACM Press/Addison-Wesley, 1998.

[6] F. Kon, “Automatic Configuration of Component-Based Distributed Systems”. PhD Thesis. University of Illinois at Urbana-Champaign, May, 2000.

[7] N. Bencomo, P. Grace, C. Flores, D. Hughes, G. Blair, “Genie: Supporting the Model Driven Development of Reflective, Component-based Adaptive Systems”, Formal Research Demonstration, In Proceeding of the 30th International Conference on Software Engineering (ICSE 2008), Leipzig, Germany, May 2008.

[8] O. Ratsimor, D. Chakraborty, A. Joshi, T. Finin, “Allia: Alliance-based Service Discovery for Ad-Hoc Environments”, In ACM Mobile Commerce Workshop, pp. 1 – 9, Atlanta, Georgia, USA, 2002.

[9] D. Chakraborty, A. Joshi, Y. Yesha, T. Finin, “GSD: A Novel Group-based Service Discovery Protocol for MANETS”, In 4th IEEE Conference on Mobile and Wireless Communications Networks, Stockholm, Sweden, 2002.

[10] F. Sailhan, V. Issarny, “Scalable Service Discovery For MANET”, In Proceedings of the Third IEEE International Conference on Pervasive Computing and Communications, pp. 235 – 244, Washington, DC, USA, 2005.

[11] C. Flores Cortes, G. Blair, P. Grace, “An Adaptive Middleware to Overcome Service Discovery Heterogeneity in Mobile Ad Hoc Environments”, IEEE Distributed Systems Online, July 2007.

[12] E. Bruneton, T. Coupaye, J. B. Stefani, “Recursive and dynamic software composition with sharing”, In Proceedings of the 7th ECOOP International Workshop on Component-Oriented Programming, Malaga, Spain, June 2002.

[13] G. S. Blair, G. Coulson, A. Andersen, L. Blair, M. Clarke, F. Costa, H. Duran-Limon, T. Fitzpatrick, L. Johnston, R. Moreira, N. Parlavantzas, K. Saikoski, “The Design and Implementation of OpenORB v2”, IEEE DS Online, Special Issue on Reflective Middleware, Vol. 2, No. 6, 2001.

[14] F. Kon, “Automatic Configuration of Component-Based Distributed Systems”, PhD Thesis, University of Illinois at Urbana-Champaign, May, 2000.

[15] J. Dowling, “The Decentralised Coordination of Self-Adaptive Components for Autonomic Distributed Systems”, PhD Thesis, Trinity College, Dublin, 2004.

[16] N. Janssens, S. Michiels, T. Holvoet, R. Verbaeten, “NeCoMan: Middleware for Safe Distributed Service Deployment in Programmable Networks”, In Proceedings of Middleware 2004, Toronto, Canada, 2004.

[17] E. Truyen, N. Janssens, F. Sanen, W. Joosen, “Support for Distributed Adaptations in Aspect-Oriented Middleware”, In Proceedings of the 7th International Conference on Aspect-Oriented Software Development (AOSD 2008), 2008.

[18] J. Silva, M. Endler, F. Kon, “Developing Adaptive Distributed Applications: a Framework Overview and Experimental Results”, Proceedings of the International Symposium on Distributed Objects and Applications (DOA), LNCS 2888, pp.1275-1291. Catania, Sicily, Italy, November, 2003.

[19] V. Issarny, M. Caporuscio, N. Georgantas, “A Perspective on the Future of Middleware-based Software Engineering”, In Future of Software Engineering 2007 (FOSE) at ICSE (International Conference on Software Engineering), Minneapolis, MN, May 2007.

[20] J. Kramer and J. Magee, “Self-Managed Systems: an Architectural Challenge”, In Future of Software Engineering 2007 (FOSE) at ICSE (International Conference on Software Engineering), Minneapolis, MN, May 2007.

[21] P. Oreizy, N. Medvidovic, R. Taylor, “Runtime software adaptation: framework, approaches, and styles”, In 30th International Conference on Software Engineering, Leipzig, Germany, May 2008.

[22] R. Rouvoy, F. Eliassen, J. Floch, S. Hallsteinsen, E. Stav, “Composing Components and Services using a Planning-based Adaptation Middleware”, In 7th International Symposium on Software Composition, pp. 52–67, Budapest, Hungary, March 2008.

[23] K. Geihs, M. U. Khan, R. Reichle, A. Solberg, S. Hallsteinsen, S. Merral, “Modeling of component-based adaptive distributed applications”, In Proceedings of the ACM Symposium on Applied Computing, Dijon, France, April 2006.