

Policy-Based Autonomic Control Service

N. Badr
School of Computing and
Mathematical Science,
Liverpool John Moores
University,
Byrom Street, Liverpool
L3 3AF, UK
cmsnbadr@livjm.ac.uk

A. Taleb-Bendiab
School of Computing and
Mathematical Science,
Liverpool John Moores
University,
Byrom Street, Liverpool
L3 3AF, UK
A.Talebendiab@livjm.ac.uk
[uk](#)

D. Reilly
School of Computing and
Mathematical Science,
Liverpool John Moores
University,
Byrom Street, Liverpool
L3 3AF, UK
d.reilly@livjm.ac.uk

Abstract

Recently, there has been a considerable interest in both policy-based management and autonomic-based management for distributed systems. However, the dynamic unpredictable nature of distributed systems makes it difficult to predict the resolution strategy at runtime. This suggests, a need to develop a policy-based autonomic control service, which has the capabilities to perform system reconfigurations with minimum intervention from human users and without having to recode the software, shutdown, or disturb the system at runtime.

In this paper we propose a control service, with autonomic capabilities, that is orchestrated by its policies (internal or external policies). The autonomic capabilities include: self-detection, self-diagnosis, self-repair, and self-reconfiguration, which may be used to stabilize/correct the system in the event of conflicts/failures. In addition, the control service provides capabilities to generate a dynamic repair strategy based on an extension to the Beliefs, Desires and Intension (BDI) model referred to as Extensible BDI (EBDI).

1. Introduction

Recently, there has been considerable interest in both policy-based management and autonomic distributed systems management [1-9]. Policies are a crucial factor that may affect the autonomic manager, which self-controls the distributed application services at runtime over a network. The dynamic

interconnectivity and the unpredictable environment of distributed systems makes it difficult to either predict the required control resolution strategy in the case of conflicts/failure, or embed static control policies in the management process.

In this paper, we describe a proposed policy-based model for evolving a self-managed, autonomic control service with minimum intervention from human users at runtime over a network. More particularly, through the integration and coordination of three main services (*service manager* and *system controller* and *JavaSpaces*) policies are used to evaluate the management process sequences.

Our approach makes use of two types of policies required for developing the autonomic control service: *external* policies, which are not attached to any particular strategy, but are used for continuous monitoring purposes and *internal* policies, which are embedded within a control repair strategy to monitor the recursion of repair actions or attributes.

In a distributed application's environment there is a need to develop an autonomic control service to support the development and lifetime management. A distributed application, which could be composed (assembled) of networked software services, may be provided with a range of deliberative capabilities. Such capabilities may include: self-governance, self-monitoring and self-repair. These capabilities may be used to enable safe predictable self-adaptation and guarantee that the required functional and non-functional properties are within the specified policies.

The autonomic control service may be used to self-detect, self-repair, and self reconfigure runtime conflicts that may occur in the distributed application

environment. In addition, the control service ensures that the system responsiveness, to its policies, is within its desirable boundaries.

The remainder of the paper is structured as follows: Section 2 provides background material relating to our approach and in particular, describes current trends in autonomic-based management and policy-based management approaches. Section 3 provides an overview of the policy-based autonomic control service. Section 4 describes the development of the control service based on the Jini middleware technology. Section 5 considers a case study and Section 6 evaluates the proposed control service based on the case study. Section 7 draws overall conclusions and mentions future work.

2. Background

Traditional software lifetime management, including software evolution activities, has been mostly conducted at maintenance-time. This requires the shutdown of a software system (or a large part of it) to enable software engineers to undertake the required maintenance, update and/or evolution of the software system.

Consequently, there is a need for enhanced methods and techniques to support the runtime management and self-governance of distributed applications. To gain an insight into these methods and techniques, we review related work to dynamic management approaches and fundamentals of software self-governance in order to develop our proposed approach.

2.1 Policy-Based Management

For the dynamic management of large distributed systems, much research has addressed the use of policies [3]. Moffett *et al.* [3] stated the necessity of representing and manipulating the policy management of distributed systems, where policy can be defined as "... *the plans of an organization to achieve its objectives ...*" [3]. In policy management, action policies are represented as a policy hierarchy, where each policy in the hierarchy represents a plan that meets a specified objective. However, the separation of policy management from the policy interpreter (i.e. managers) facilitates both the dynamic change in the distributed system management process and the reuse of managers in different processes [10].

However, there is a need for approaches to program network components and policies specifications. Sloman and Lupu [11] studied authorization and

obligation policies' specification for programmable networks. Such policies are interpreted to facilitate runtime activation, de-activation and/or their modification without having to shut down the network node. Policies could be grouped to refer to the same subject domain and to propagate to the assigned managers [1].

Other approaches to policy-based management have used condition-action rules to support a static policy configuration-based solution, in which human intervention is required for system reconfiguration and policy deployment. However, Moffett *et al.* [12] have proposed a framework for supporting automated policy deployment and flexible event triggers to permit dynamic policy configuration, focusing on solutions for dynamic adaptation of policies in response to changes within the managed environment.

Other research efforts have focused on policy specification and enforcement for dynamic service management. For instance, the IETF Policy Working Group is developing a QoS network management framework using the X.500 directory schema [13]. IETF policies are encoded as If-Then conditions and stored in directories. The component-policy mapping is done by interface roles. The IETF policy group has also focused on the network layer and not the application layer. Yoshihara *et al.* [4] have proposed a framework that adapts policy parameters for network monitoring. In the framework, a management script is expressed using the IETF Policy Working Group's proposed representation. This representation encompasses: policy specifications, policy management life-cycle, system's notification related to QoS threshold violations and prototyped using the differentiated services. Similarly, Brunner *et al.* [14] have addressed system design for managing QoS in Multi-Protocol Label Switching (MPLS) networks by extending the Common Information Model (CIM) policy model with MPLS specific classes

2.2 Autonomic-Based Management

Within the distributed system community there is a need to design and build computing systems capable of running themselves, adjusting to unpredictable changes and handling resources efficiently [15]. The two main elements of autonomic management are the *functional unit* and the *management unit*. The functional unit performs the main operation and is provided by elements such as web services or databases, etc. The management unit is responsible for system resources and operational performance and

hence the reconfiguration of resources according to adaptive changes [5]. Autonomic systems have been defined by IBM [6] as system that have” ...*The ability to manage themselves and dynamically adapts to change in accordance with policies and objectives ...*”.

Intel research is exploring the paradigm of proactive computing. Proactive computing has similar aims to those of autonomic computing, but the main difference is that autonomic computing focus on managing the computing system complexity, whereas proactive computing considers the need to monitor and build complex real-world interactions [16]. Autonomic computing systems have the ability to monitor, diagnose and heal themselves. This requires that the system has the ability to dynamically insert and remove code in real-time systems, a technique known as “hot swapping”.

Hot swapping [9] is proposed as a means to enable autonomic software systems by either *interposition* or *replacement* of the code. Interposition involves inserting a new component between two existing ones. For example, inserting one monitoring component when a failure is detected at runtime. Replacement allows an old component to be “swapped” with a different implementation while the system is running. The new component may then continue with the management of resources. However, successful autonomic systems not only need to self-detect, self-diagnose, self-heal, but they must also self-protect to allow autonomic management in a secure environment [5].

3. Policy-Based Autonomic Control Service

Our approach [17-19] extends the ideas described previously in the related work by integrating the autonomic control service within a policy-based service in order to turn “control at design time” into “control at runtime” [9]. In addition the approach proposes a new mechanism that can add to the management of distributed applications in a flexible manner. This mechanism allows external reusable autonomy, namely the EBDI model, to find the appropriate resolution strategy in the case of runtime failures/conflicts. The requirements for managing distributed applications start by providing the system with abilities to detect, classify, fix and reconfigure at runtime *without* any disturbance and/or system shutdown and *with* minimum intervention from human users. Runtime conflicts and systems errors are typically difficult to predict, detect and rectify at

runtime without introducing a new approach that dynamically rectifies the inconsistencies or failures

The essential requirements for developing the autonomic (self-control) runtime management of distributed applications are addressed by the three main, mentioned previously. The three services are integrated to evolve the proposed control service as will be considered further in Section 4. The requirements to be addressed are categorized according to different autonomic concerns, including the following:

- Conflict self-detection, where the monitoring model in our control service is responsible for conflict detection in either a decentralized fashion, as in a service-monitoring model (*service manager*), or centralized, as in a system-monitoring model (*system controller*).
- Conflict self-diagnosis, which includes: conflict identification and conflict classification that provides the basis for the solution and repair strategies that trigger the appropriate action for such conflicts.
- Conflict resolution and self-repair strategies, which address dynamic and generic selection, execution and evaluation phases.
- System dynamic adaptation, reconfiguration and coordination among the services in a distributed computing environment.
- Control policy base that includes two types of policies:
 1. External control policies: separating the policies from the manager where policies are not attached to any particular strategy [20], but are used for continuous monitoring purposes, such as repair, task/execution progress monitoring and coordination
 2. Internal control policies: which are embedded within a resolution strategy to monitor the recursion of a strategy’s actions or attributes (i.e. partial solutions).

Figure 1 below, describes the main requirements and the interaction process. This starts with the monitoring process to detect conflicts using the control service internal and

external policies. This followed by the diagnosis process and then the repair strategies, which are based on the proposed EBDI model (considered further in Section 4). The strategies are embedded in an external format such as XML, which may be parsed and translated into executable format by the system interpreter. In addition the distributed shared space is used to facilitate the remote integration and coordination of the distributed control services.

4.The Development of The Proposed Control Service.

The development of the proposed policy-based autonomic control service combines three main services: the *Service Manager*, the *JavaSpace Service* and the *System Controller*.

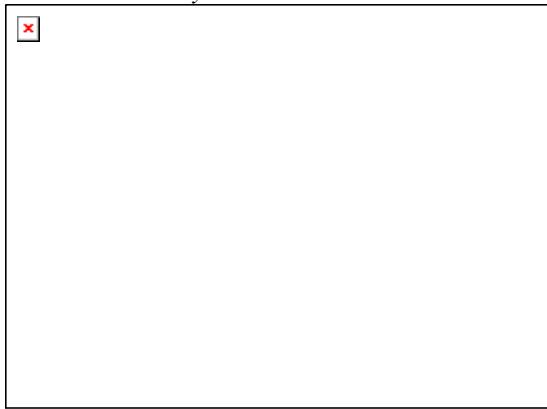


Figure 1: The autonomic management processes requirements.

The three main services are embedded in a three-layered design model, shown in Figure 2. The three-layered model comprises: the middleware core services (Sec. 4.1), the autonomic control service (Sec. 4.2), and user applications services, which are regarded as a federation of services distributed over a network. This architecture is based on a control service model that follows a cycle of monitoring the target application, detecting undesirable behaviours, identifying conflicts/errors, prescribing remedial action plans and enacting change plans through reconfiguration.

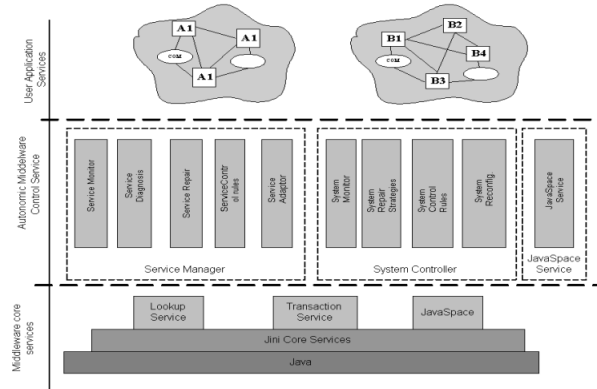


Figure 2: The control service architectural layers.

4.1 The Middleware Core Services Layer

The middleware core services contain three basic services:

- The registration service, this service is responsible for registering the application service's object, which is the visible part of the service that will be downloaded by clients over the network (e.g. Jini lookup's protocol). Typically, the registration service will "listen" on a dedicated port for registration requests.
- The lookup service, this service is responsible for locating registered services. The clients establish a request using a template, which is compared with the service proxies that are currently stored on the network. If a matching proxy is found, then a copy of the matched service proxy is moved from the network to the client machine
- The distributed shared memory which is based on a persistent object model used to store, exchange and coordinate the activities of interacting distributed processes. The processes communicate indirectly by exchanging objects via the shared spaces.

4.2 The Autonomic Control Service Layer

The control service incorporates three main services and their interactions are shown in Figure 3. The three main services are: the service manager, JavaSpace and the system controller. The three main services are used to evolve the autonomic control service, based on the requirements summarized in section 3.

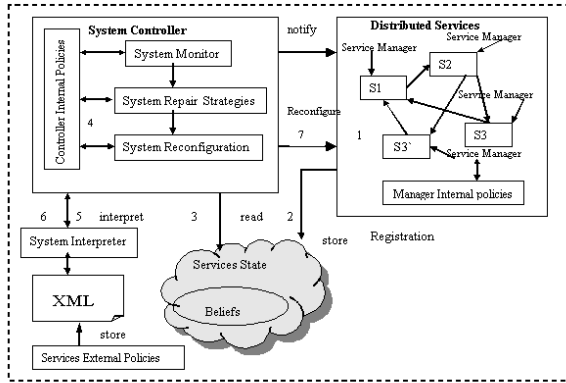


Figure 3: The autonomic middleware control service architecture.

Figure 3, outlines the sequence of the control process, which is described below:

1. Each service has an assigned service manager that is activated when the client requests a service. The manager “looks after” its application-level service and reports the service status to the system controller either directly (e.g. by remote event) or indirectly (e.g. by the distributed shared space).
2. The service manager posts both a service state and a Jini lease object reference for its service to the shared space. The space may be accessed by the system controller, or other service managers to determine the application-level service’s state.
3. Typically, the system controller will monitor the shared space at regular intervals to check the workflow and for conflicts in service states, as reported by its manager.
4. The system controller detects and repairs conflicts and then reconfigures the system by forwarding a repair decision to the system manager.
5. An externalized dynamic document for the repair strategy (i.e. an XML document) is provided to provide the normative intended resolution (i.e. EBDI sec.4.2.3) for a conflict.
6. A system interpreter translates the external format of the strategy document to an executable format.
7. A reconfiguration process reconfigures the whole system by adapting its structure/behaviour according to the new repair strategy.

The three main services (service manager, system controller and the JavaSpace service), which combine to enable reconfiguration, are described further below.

4.2.1 The Service Manager

Service controller or manager concepts have recently gained popularity amongst the autonomic community as typified by [20]. In this work, the service manager is used to adapt the structural components and the dynamic behaviour of the services they provide. Structural components can evaluate their behaviour and environment against their specified goals with capabilities to revise their structure and behaviour accordingly [20]. Our choice of using a service manager for each individual application-level service provides the main separation of concerns to our control service. This separation into distinct service managers eases their management by decentralizing the control of each individual application-level service. Service managers look after their application-level service and monitor its behaviour using an external and internal policies. The service manager incorporates four main elements:

- Service Monitor, which is an essential utility of service self-management and the adaptation process. The monitor possesses the ability to monitor its service’s behaviour using service internal policies that check the service low level attributes and parameters at run-time.
- Service Diagnosis, which identifies the failure type that is essential to accurately target the root cause of errors and to initiate the appropriate remedial plans, whilst preventing any further system errors.
- Service Internal Policies, which are used to measure the service parameters before selecting the resolution or repair operators that are activated by the repair process to allow any required alterations. In addition, the internal policy checks the execution of each process of the service manager (i.e. Service monitoring, diagnosis, repair, and adaptor).
- Service Repair, which provides operators to resolve conflicts. Self-repair selects the required operators and associated operations that support dynamic changes to an application-level service during its lifetime. The operators essentially identify the primitive corrective operations used in a repair strategy. If a repair should fail a set of exception handling classes are used to coerce the safe termination of a

system that has failed. Typically, operators provide primitive operations for adding and removing components and connections. However, certain processes can provide operators with a higher level of abstraction.

- Service Adaptor, which is based on the use of a parametric adaptation loop construct, where the service adaptor receives messages or notification from the service repair process. In such cases, the service adaptor may determine the changeable parameters and attributes of the application-level service for updating/changing during the system's lifetime. Alternatively, the service adaptor may catch exceptions and feedback the result to close the loop for monitoring its application-level service's behaviour over again.

4.2.2 The JavaSpace Service

The JavaSpace service is a persistent service based on distributed tuples that provides the autonomic control service with a distributed shared memory/space. This shared space is used to coordinate the relationship of shared resources or services in the distributed application over the network. In addition it has the capability to store required information and report service states for use by the system controller service to repair and reconfigure the system in the event of any service failures. The JavaSpace service uses *external policies* to notify the system controller directly with remote events (i.e. through remote event notification). The remote events may either notify of changes in application-level service states, reported by the service manager, or notify the system controller if the lease of a service manager should expire. Figure 4 shows how the monitor can read the JavaSpace service to check the application-level service's state.

```

BasicEntry service_state =null,
BasicEntry serviceTemplate= new BasicEntry();
JavaSpace space = SpaceAccessor.getSpace(
);
BasicEntry myService = new BasicEntry (str,0);
try{
    service_state = (BasicEntry) space.read
    (template, null, Long maxValue);
} catch (remoteException rem_exp)
{ rem_exp.printStackTrace(); }

```

Figure 4: The monitoring of the service state from the JavaSpace

4.2.3 The System Controller

The system controller is responsible for the dynamic resolution/reconfiguration of the distributed application as a whole. The controller is also responsible for coordinating the activities of individual service managers. The individual service managers are used to monitor, repair and adapt their associated application-level services in the event of conflicts/failures. Service managers may then report their associated application-level service states using the JavaSpace service. The external policies are used to either advocate the JavaSpace notification to the system controller or for the activation of the system controller to initiate the monitoring of the application-level service states, reported by their service managers.

The system controller service contains three main components: a system monitor, a system repair strategy process, and a system reconfiguration process. These components, which are shown in Figs. 2 and 3, are explained further below:

- The System Monitor has the ability to collect or receive information that is required to support and guide the resolution strategies within the control process using the external policies. Bearing in mind the dynamic nature of distributed services, the system controller cannot choose suitable strategies and execute appropriate actions without continuous monitoring of application services, via their service managers. Such system monitoring (i.e. self-detection) requires continuous feedback processes that are supported by the use of the external policies to activate the feedback processes.
- System Repair Strategies are an essential aspect of autonomic control service. Autonomic computing requires distinct forms of strategies, which consist of resolution actions, using the *external policies* to monitor and orchestrate the execution of the strategy actions. As an example, the action shown below may be used if the average latency of the control process exceeds the maximum allowed latency. If such a case occurs, the control rule detects and fires an execution failure.

```

If (control_avLatency is larger than
    control_maxLatency)
start
    conflict_monitor(true);
    start_control_process();
end;

```

A strategy determines *when, where* and *how* the repair and subsequent adaptation is applied. Repair strategies must consider the functions of the application and its services, the operating environment and its attributes and properties. Our approach to repair strategies relies on a dynamic solution approach, which uses XML documents. A lightweight system interpreter is used to translate the XML resolution strategies into executable operations. The interpreter is used to map and bind the XML-encoded “tags” of a repair strategy to Java methods. This process is implemented using the Java reflection API [21], which allows the Java methods to be invoked based on their XML encoded counterparts.

The resolution strategies are used to represent the effect of various alternatives resolutions based on BDI model that represent the Beliefs, Desires and Intension. However, the BDI model has a lack of policies and norms that are essential for evaluating the resolution strategies (i.e. Intention). Various extensions to the BDI model were proposed to address some of its documented weaknesses, including a normative model namely; EDA [22], and BOID [23]. However, these extensions do not address the usage of these norms and policies for optimizing its selected repair strategies (i.e. intentions).

The development of system repair strategies module was based on our proposed Extensible Beliefs, Desires and Intension (EBDI) model [19]. The EBDI model is a policy based model that provides a highly suitable architecture for the design of situated intentional software that continuously monitors and/or observes its environment and acts to change in accordance with its situated BDI, which is normative and policy based model for selecting the appropriate intension or strategy.

- Beliefs, which correspond to service information derived and/or accessed from a range of sources, including domain, environment or beliefs of other services.
- Desires, which represent the state of affairs (i.e. in an ideal world), which often maximize the service’s own goals.

By comparing a system beliefs set (observed system states) against its desires the system may detect a mismatch and trigger intentions (instantiate a set of intentions) [24].

- Situated intentions, which represent action sets for the system to undertake in a given situation to achieve its specified desires and/or to address the mismatch between the system environment (beliefs) and the system’s desires (goals).
- Normative intention, which represents a set of actions to be undertaken to ensure a specified set of external policies. The external policies, which may include obligation and responsibility rules, are observed before a given intention is enacted and/or affective rules, emerging as a result of an enacted intentions set, are followed. For example, the intention strategy may be to remove a client, but the external policy may specify:
`all_client.important=high.`
- Utility intention, which represents a set of system actions examined by the *external policies* to optimize its goal-oriented intentions, such as cost or QoS.
- System Reconfiguration to achieve “optimal” reconfiguration, the controller takes into account a number of considerations to avoid any further downstream inconsistencies, including performance degradation and/or failures. Similar to [25], we defined a set of reconfiguration operators required to perform an application software architectural transformation. These operators were encoded in external XML document format. Typical operators are:
`getServiceManager(), getClient(),`
`and notifyClient().`

5. Case Study

The policy-based autonomic control service has been demonstrated through an industrial case-study, namely EmergeITS[19]. EmergeITS (Fig. 4) is a software simulation that was developed using Jini middleware technology to demonstrate the concept of intelligent networked vehicles. EmergeITS was developed and prototyped in collaboration with the Merseyside Emergency Fire Service.

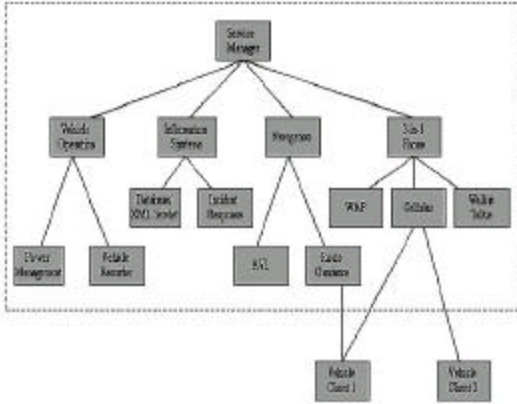


Figure 5: The Architectural view of the EmergeITS application.

One crucial service of the EmergeITS is the 3-in-1 phone service, which allows a mobile phone or PDA device to be used in one of three different modes: a cellular phone, a WAP phone or a walkie-talkie. The 3-in-1 phone may be used for either voice communication or to receive multimedia content, subject to the requirements of the user and availability of a communication service provider. An autonomic middleware control service prototype was incorporated into the case study to provide the meta-control software to support EmergeITS over the network.

The 3-in-1phone service is hosted by an in-vehicle computer, which also acts as a gateway. If the local service deployment should fail, then a service manager is started to provide self-monitoring, self-diagnosis, self-repair and self-adaptation. In the event of failure, the service manager will select an appropriate repair strategy, thereby providing a degree of conflict resolution and fault tolerance. The service manager stores the 3-in-1 phone service state in a JavaSpace service and the system controller may check these states against external policies. The sequence of processes of the autonomic control service proceed as follows:

- The service manager monitors the method invocations made by clients, such as `connect()`, `disconnect()`, and `send()`, `receive()`. If an invocation should fail the service manager will try to repair the failure using its repair operators (e.g. `notify()`, `add_Client()`, `remove_Client()`).
- There are two main factors used by the internal policies to determine the selection of the operator for specific conflicts. The first factor is the satisfaction of the service's attributes and

parameters. The second factor is the execution of the operator without exceptions, bearing in mind the runtime systems current structure and behaviour. Examples of repair operators are: `remove_Client()`, which may be invoked by the service manager to remove or disconnect a client who had low priority and has been connected for more than a predefined time. In such a case, the client would be removed from the list of clients that are using the service as soon as the service's policies/rules are checked. The internal policy to enforce this is:

`service.num_connections` is larger than
`service.max_connections`

where `num_connections` is the current number of connected clients and `max_connections` is the maximum number of connected clients.

- The service manager then informs the system controller service of the state of the 3-in-1 phone service, either by direct notification, or via the JavaSpace service. The latter case requires that the system controller service read the JavaSpace service at regular intervals. These communications, made by the service manager to the system controller, are activated by external policies.
- If further exceptional behaviour prevails, the external policies of the system controller service will trigger a monitor service to initiate a conflict resolution process and subsequent reconfiguration as appropriate. For example, in the 3-in-1 the invocation of the `connect()` method on the GSM service, may result in a `RemoteConnectionException` due to the unavailability of a GSM service.
- This exception is then checked by the system controller service, resulting in the activation of a suitable system resolution strategy based on the EBDI model. For instance, the repair strategy may first attempt a specified number of connection retries. If the retries are unsuccessful, the strategy then searches for an alternative GSM service provider or connects to another service such as WAP. Figure 5 below, shows an example of an EBDI-based conflict repair strategy encoded as an XML document.

```

<?xml version="1.0" ?>
- <!-- Simple Description of 3in1 phone Strategies
-->
<!DOCTYPE strategy (View Source for full doctype...)>
- <Strategies>
- <Strategy id="1" type="desire">
- <Action id="1" type="plan" name="Connection">
- <Properties>
- <property id="1" name="host">cmnbnadr</property>
- <property id="2" name="Location">GPS_loc</property>
- <property id="3" name="Max_connected">maxNo</property>
- <property id="4" name="Method">connect</property>
- </Properties>
- </Action>
- </Strategy>
- <Strategy id="2" type="intension">
- <Action id="1" type="plan" name="Retry">
- <Properties>
- <property id="1" name="No_trial">two</property>
- <property id="2" name="serviceStatus">not null</property>
- <property id="3" name="Method">connect</property>
- </Properties>
- </Action>
- <Action id="2" type="plan" name="Alternative">
- <Properties>
- <property id="1" name="host">cmnbnadr</property>
- <property id="2" name="New Manager Interface">ManagerProxy</property>
- <property id="3" name="Get Manager">getServiceManager</property>
- <property id="4" name="Client Interface">ClientProxy</property>
- <property id="5" name="Get Client">getClient</property>
- <property id="6" name="Notify Client">notifyClient</property>
- <property id="8" name="Max_connected">maxNo</property>
- <property id="9" name="Method">connect</property>
- </Properties>
- </Action>
- </Strategy>
- </Strategies>

```

Figure 6: The XML description of a self-repair strategy.

6. Evaluation

The evaluation considers the 3-in-1 phone case-study and compares performance with and without the autonomic middleware control service. The elapsed time used to undertake a control process was used as a quantitative metric to indicate the applications performance profile with and without the autonomic control service.

In addition, a set of evaluation metrics often used for control systems were applied to provide a general guide for evaluating the autonomic control service. The evaluation metrics are Stability, Robustness, Performance Profile, and Average Latency, where the average latency is used to measure the average time required for the controller to start up its control cycle

In order to start the evaluation it is necessary to define the control requirements for the autonomic control service. Such control requirements encompasses a range of applications and/or domain knowledge including:

- Control policies, which provide a boundary range to examine runtime conflicts and to check whether the system is stable and robust or not. For example an external policy may specify:

average_latency is less than max_latency

where the average_latency is the elapsed time the control process has already taken to achieve its control process, and max_latency is the maximum elapsed time the control process should take to achieve its control process.

- EBDI-based repair strategies and its associated utility function to underpin the system decision-making processes when evaluating different adaptation plans. For example, to select and optimise a repair strategy (repair plans) from a large search space of possible repair plans.

Here we will use the elapsed time of the whole process and the average latency of the control process itself. The quantitative evaluation results for the 3-in-1 phone, shown in Figs. 7, 8 and 9, are considered in the following system situation:

- Without conflict occurrence and without control services.
- With conflict occurrence and without control services.
- Without conflict occurrence and with control services.
- With conflict occurrence and with control services.

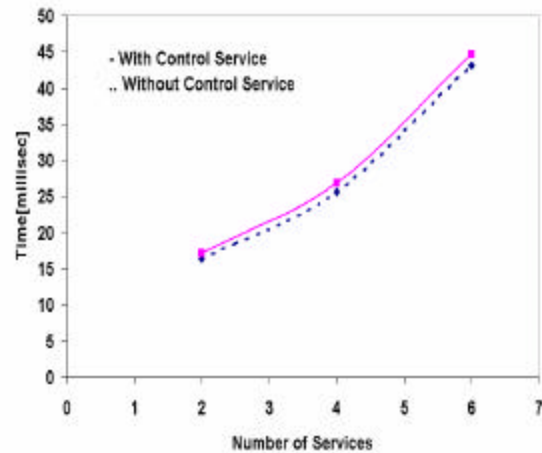


Figure 7: Comparison of the elapsed time with and without the autonomic middleware control service without conflict occurrence.

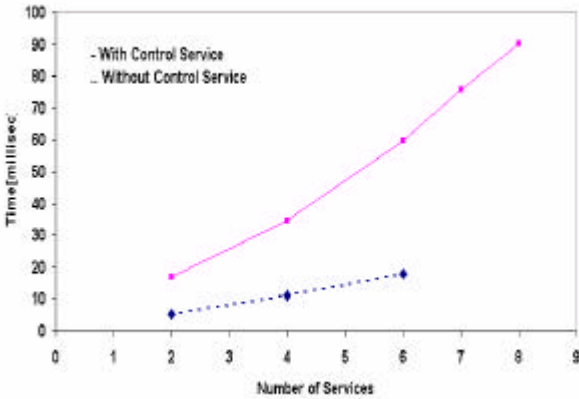


Figure 8: Comparison of the elapsed time with and without the autonomic middleware control service with conflict occurrence.

Figs. 7 and 8 represent the elapsed time (y-axis) against the number of services (x-axis). The solid line represents the system with the policy-based autonomic control service while the dotted line represents the system without the policy-based autonomic control service.

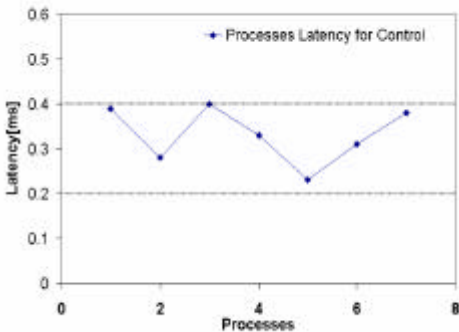


Figure 9: Average latency while running the control process for the 3in1 phone example.

Figure 7 shows that both curves are very close for the case without any conflict occurrences. However, in Figure 8 the two curves are forced further apart when conflicts occur. The reasons for this are explained as follows:

- The results show that the system consumes more time in the conflict occurrence situation. However, it is possible to tune autonomic control services to an average latency, such as that of Fig. 9. If an autonomic control service does not complete within this time, there may

still be an outstanding conflict, but this can be solved as part of the feedback process.

- The average latency is measured so that it may be checked within external policy against the maximum latency. Otherwise the controller's feedback loop will continue so that the conflict may be corrected in the next-pass. As a result of the experimental measurements, the maximum average latency experienced should not exceed that of Fig. 9.
- As can be seen from Fig. 8, the curve without autonomic control (dotted line) halts as soon as catastrophic failure occurs, whereas the curve with autonomic control (solid line) proceeds in the event of such failures.

7. Conclusions and Future Work

In this paper we have described the development of a policy-based autonomic control service. The control service incorporates the policies (either internal or external) within the architecture of the autonomic control service to achieve policy-based autonomic control of distributed applications at run time. The control service was implemented using Jini middleware and as such extends Jini's capabilities to provide an autonomic control service.

We have described how the proposed extensible BDI (EBDI), which is a policy (normative)-based model that is used to achieve in the autonomic control service. In particular, EBDI is used to examine the system external policies and for generating the appropriate repair strategy at runtime.

The main services that are combined to form our policy-based autonomic control service are: service manager, JavaSpace service and system controller service. We have described how these services may be used to control a 3-in-1 phone application service. In addition we have evaluated the performance of the autonomic control service using the elapsed time and the average latency as metrics of the system at runtime.

In our future work, we intend to improve/enhance the autonomic control service on two main fronts: self-protection and machine learning. There is a need to provide self-protection to provide security in untrustworthy environments. There is also a need to equip the current model with the machine learning capabilities so that the policy-based autonomic control service can access history and knowledge relating to the previous failure cases.

8. References

1. E. Lupu, D. Marriott, M. Sloman, and N. Yialelis. *A Policy Based Role Framework for Access Control*. in *First ACM/NIST Workshop on Role-Based Access Control*. 1995. Maryland USA: ACM.
2. E. Lupu and M. Sloman, *Conflicts in Policy-based Distributed Systems Management*. IEEE Transactions on Software Engineering, 1999. **25**(Special Issue on Inconsistency Management): p. 852-869.
3. J. Moffett and M. Sloman, *Policy Hierarchies for Distributed Systems Management*. IEEE Journal on Selected Areas in Communications, 1993. **11**: p. 1404-1414.
4. K. Yoshihara, M. Isomura, and H. Horiuchi. *Distributed Policy-based Management Enabling Policy Adaptation on Monitoring using Active Network Technology*. in *12th IFIP/IEEE International Workshop on Distributed Systems: Operations and Management*. 2001. Nancy France.
5. D. Chess, C. Palmer, and S. White, *Security an autonomic computing environment*. IBM SYSTEMS JOURNAL, 2003. **42**.
6. IBM, *Autonomic Computing*.
7. IBM, *Autonomic Computing Concepts*.
8. IBM-Research, *Applications of Multi-Agents Learning in E-Commerce and Autonomic Computing*.
9. J.Appavoo, et al., *Enabling autonomic behavior in systems software with hot swapping*. IBM SYSTEMS JOURNAL, 2003. **42**.
10. M. Sloman, *Policy Driven Management for Distributed Systems*. Journal of Network and Systems Management, 1994. **2**.
11. M. Sloman and E. Lupu. *Policy Specification for Programmable Networks*. in *First International Working Conference on Active Networks (IWAN'99)*. 1999. Berlin.
12. L.Lymeropoulos, E.Lupu, and M.Sloman. *An Adaptive Policy Based Management Framework for Differentiated Services Networks*. in *Proc. 3rd IEEE Workshop on Policies for Distributed Systems and Networks (Policy 2002)*. June 2002. Monterey, California.
13. K. Barber, T. Liu, H. Goel, and C. Martin.. *Conflict Representation and Classification in a Domain Independent Conflict Management Framework*. in *the Third International Conference on Autonomous Agents*. 1999. Seattle WA.
14. M. Brunner and J. Quittek.. *MPLS Management using Policies*. in *Proc. IM 2001: 2001 IEEE/IFIP International Symposium on Integrated Network Management*. 2001. Seattle USA.
15. P. Horn, *Autonomic Computing: IBM's Perspective on the State of Information Technology*. IBM Corporation, 2001.
16. R. Want, T. Pering, and D. Tennenhouse, *Comparing autonomic and proactive computing*. IBM SYSTEMS JOURNAL, 2003. **42**.
17. D. Reilly, A. Taleb-Bendiab, A. Laws, and N. Badr. *An Instrumentation and Control-Based Approach for Distributed Application Management and Adaptation*. in *Workshop on Self-Healing Systems (WOSS'02)*. 2002. Charleston SC USA.
18. N. Badr, D. Reilly, and A. Taleb-Bendiab. *A Conflict Resolution Control Architecture for Self-Adaptive Software*. in *the International Workshop on Architecting Dependable Systems: WADS 2002 (ICSE 2002)*. 2002. Florida USA.
19. N. Badr, *An Investigation into Autonomic Middleware Control Service to Support Distributed Self-Adaptive Software*. PhD Thesis. School of Computing and Mathematical Sciences. Liverpool John Moores University. Liverpool. 2003
20. E. Lupu and M. Sloman. *Conflict Analysis for Management Policies*. in *Fifth IFIP/IEEE International Symposium on Integrated Network Management IM'97*. 1997. San-Diego: Chapman & Hall.
21. Sun-Microsystems, *Trail: The Reflection API*.
22. J. Filipe. *A Normative and Intentional Agent Model for Organisation Modelling*. in *Third International Workshop in Engineering Societies in the Agents World*. 2002. Madrid Spain.
23. *Home of the BOID*.
24. M.Bratman, *Intentions, Plans, and Practical Reason*. Harvard University Press, 1987.
25. D.Garlan and R.Stratton. *Architecture-based Adaptation of Complex Systems*. accessed January 2002.