# An Instrumentation and Control-Based Approach for Distributed Application Management and Adaptation

D. Reilly, A. Taleb-Bendiab, A. Laws, N. Badr
Liverpool John Moores University
Byrom Street
Liverpool, L3 3AF
044 0151 231 2489

{d.reilly; a.talebbendiab; a.laws; cmsnbadr}@livjm.ac.uk

## ABSTRACT

Distributed applications are notoriously difficult to develop and manage due to their inherent dynamics and heterogeneity of component technologies and network protocols. Middleware technologies dramatically simplify the development of distributed applications, but they still prove difficult to manage at runtime. This paper considers the "on-going" development of a framework that provides instrumentation and control services, which extend core middleware services, to realize the runtime management and adaptation of distributed applications. The instrumentation and control services are used in conjunction with dependency management utilities to measure performance, monitor behaviour and resolve the runtime inconsistencies and conflicts that may occur in distributed applications.

## Categories and Subject Descriptors

D.3.3 [**Programming Languages**]: Language Constructs and Features – *classes and objects, control structures, frameworks.*

## General Terms

Design, Management, Experimentation, Languages.

## Keywords

Middleware, Instrumentation, Control, Dependency Management, Jini Technology.

## 1. INTRODUCTION

Distributed component-based software applications consist of a collection of software components that communicate via a distributed middleware. Some components provide prescribed functionality as *services* to other components, which adopt the role of clients. Taken together, components operate in a peer-to-peer fashion to coordinate their activities giving the impression of a single, integrated computing facility. The distributed middleware, or simply middleware, plays a crucial role by providing APIs and support functions that effectively bridge the gap between the network operating system and distributed components and services.

The runtime management of distributed applications is difficult because of the possible use of different component technologies (DCOM, CORBA, Enterprise Java Beans (EJB)), different network protocols (TCP/IP and UDP) and the dynamic behaviour, inherent in distributed applications, that can give rise to component reconfigurations that occur "on-the-fly". These problems can lead to runtime inconsistencies and conflicts, which suggests a need for adaptation techniques to minimize or completely rule-out their effects. Such adaptation can be realized through the combination of instrumentation and control techniques, which have a proven success record in the management of conventional engineering systems.

For the combination to prove successful, where distributed applications are concerned, the instrumentation and control services need to be aware of the architecture and connectivity of components and services. In short, they need to be aware of the *dynamic dependencies* that exist between components and the services they provide. Such dependencies occur when client components discover and use services provided by other components, thereby becoming dependent on their services until they are no longer needed and can be discarded. This paper considers how instrumentation and control techniques can use a dependency model as the basis for the reasoning and decision-making processes that are required to adapt a distributed application to correct runtime inconsistencies and conflicts.

The remainder of the paper is structured as follows: section 2 provides a brief review of recent significant developments in self-adaptive software, concluding with a statement of our contribution. Section 3 provides an overview of software instrumentation, software control and dynamic dependencies, which feature as the main services in our framework. Section 4 considers the development of the framework, based on the


*Conference '00*, Month 1-2, 2000, City, State.

combination of these services. Section 5 describes a recent case study conducted to evaluate the framework. Finally, section 6 draws overall conclusions and mentions future work.

## 2. SELF-ADAPTIVE SOFTWARE

Self-adaptive software can be seen as a new architectural style, which extends the concepts of software control to adapt the structural configuration and dynamic behaviour of an application. Laddaga, [7], defines self-adaptive software as:

*"Software that evaluates and changes its own behaviour when the evaluation indicates that it has not accomplishing what it is intended to do, or when better functionality or performance is possible"*.

Such an architectural style presents an attractive concept for developing self-governing software that partially or fully accommodates its own management and adaptation activities. In line with our own approach, several researchers, active in this area, have adopted control engineering concepts. The control engineering approach is typified by Osterweil and Clarke, [10], who present an architecture that uses a controller with a well-specified control function with *feedforward* and *feedback* loops to enable a target system to be monitored to regulate its operation in accordance with its given control model.

The control theory based paradigm provides a framework for designing software that supports self-control during the operation of the software. According to Kokar *et al*., [6], the self-controlling software model supports three levels of control: feedback, reconfiguration and adaptation. Meng, [8], proposed a control system for self-adaptive software based on a descriptive model of a self-adaptive control system, which also employs the control system concepts of feedforward and feedback. In Meng's approach, the feedforward process provides specifications of the software and its behaviour and the feedback process gathers and measures the software's environmental attributes, which are used as the basis for adaptation.

Robertson *et al*. [13], discuss the similarities between self-adaptive and reflective architectures and the benefits that reflection provides, but also highlight the difference between the two architectural styles. Robertson *et al*. explain how self-adaptive software uses a model of the computation against which the current state and a desired goal state are compared and the semantics of the computation are adjusted accordingly to minimize any difference. Recently there has been an increasing research trend focused on *self-healing* and *self-repairing* software that provides the capability to modify structural and/or behavioural models at runtime. Cheng *et al.* [2], describe an approach to self-repair based on using an architectural model of the system, which adopts a particular architectural style, as a parameter in the monitoring/repairing framework.

We based our approach on several of these previous ideas, but decided to use a dependency digraph as the architectural model to provide reference points and goal states. The digraph provides a model of the distributed computation that can be used by instrumentation services to measure performance and monitor behaviour and by control services for reconfiguration and hence adaptation. The approach does however require that application components satisfy one condition, which is that they must include a reference to a dedicated *manageable proxy object*. The manageable proxy, considered further in section 4, is used primarily to represent dependency relationships, but also to provide a separation of concerns by "factoring out" the code that is referenced and used during adaptation strategies. The adaptation is performed by a middleware-based combination of instrumentation and control services that use dependency relationships as a model of the computation. Overall, the approach provides a practical yet effective solution that can be used, in conjunction with core middleware services, to facilitate the runtime management and adaptation of distributed applications.

## 3. FRAMEWORK SERVICES

Instrumentation and control services together with a dynamic dependency model form the basis of our framework. Each of these concepts are further explained below.

### 3.1 Software Instrumentation

Software instrumentation[1] has been used for some time in software engineering to debug and test software applications and also for monitoring performance and producing runtime metrics. Traditional, *static* instrumentation approaches involved the insertion of additional software constructs at design-time (via compiler directives), or when the system was off-line, during maintenance, to observe specific events and/or monitor certain parameters. Where distributed applications are concerned, the limitations of static instrumentation have led to interests in *dynamic* instrumentation that can be applied (and removed) as required at runtime. Dynamic instrumentation makes use of instruments such as gauges, probes and monitors (as used in conventional engineering) that can be dynamically attached to application components to measure specific runtime parameters and monitor their behaviour.

The potential of dynamic instrumentation has been recognized by the distributed computing community and the DARPA funded initiative for *Dynamic Assembly for System Adaptability, Dependability and Assurance (DASADA)* is a prime example of this recognition. The DASADA programme consists of several projects concerned with the use of software gauges and probes to deduce component configurations and for runtime monitoring and adaptation, as typified by Garlan *et al.* [4]. Also of interest is the work of Diakov *et al*. [3], who use reflective techniques to monitor distributed component interactions by combining CORBA's interceptor mechanism together with Java's thread API to "peek" into the implementation of CORBA components at runtime.

[1] The term "instrumentation" is used henceforth to refer to software instrumentation.

## 3.2 Software Control

Controller concepts and control services have recently gained popularity amongst the self-adaptive software community (as typified by Osterweil and Clarke in [10]) who use control services to adapt the structural configuration and dynamic behaviour of an application. Structural components can evaluate their behaviour and environment against their specified goals with capabilities to revise their structure and behaviour accordingly. Control services make use of well-specified control functions with feedforward and feedback loops to enable a target application to be monitored and hence regulate its operation in accordance with its given control model. Control services are based on the two tasks of monitoring and diagnosis:

1. The monitoring task uses of a set of control rules against which monitored behaviour and architectural configuration are checked to detect conflicts.
2. The diagnosis task involves the execution of control rules, activated by conflicts, which identify the causes of conflicts and provide the basis for the selection of conflict resolution and adaptation strategies.

As explained further in section 4, both instrumentation and control services were implemented as Jini activatable services so as to limit the overhead that they may impose on the application's middleware layer.

## 3.3 Dynamic Dependencies

Dynamic dependencies (considered further by Hasselmeyer in [5]) can be represented as the arcs or edges in a directed graph (digraph) in which components (and their services) represent the nodes. Conceptually, a dependency is a directed relationship between a set of components, as shown in Figure 1, that can be represented using the ideas of Hasselmeyer in [5] who defines the relationship based on two roles: the dependent component (a client) and the free or independent component(s) (service provider(s)).

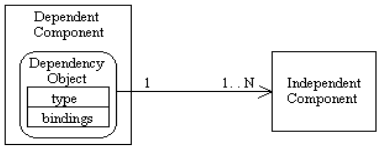


**Figure 1. Dependency conceptualization.**

Physically, we represent the directed dependencies of a dependent component using a Java `Vector` object in which each element represents a single dependency and each dependency provides information such as component bindings and the type of service provided by the independent component. Dependency vectors are associated with a dependent component, via a manageable proxy reference within the dependent component. As considered further in section 4, the manageable proxy implements a *manageable interface* through which dependency management utilities provide an API. The dependency management API allows instrumentation and control services to access individual dependency vectors or even the complete dependency digraph for the reasoning and decision-making processes prior to the reconfigurations of an adaptation strategy. Essentially the dependency digraph provides an architectural point of reference or goal state in that all dependencies must be satisfied following an adaptation strategy.

# 4. DEVELOPMENT OF THE FRAMEWORK

This section considers the "on-going" development of the framework, which has been implemented using Jini, the Java-based middleware technology developed by Sun Microsystems, [14]. Essentially Jini, together with Java's Remote Method Invocation (RMI), allows distributed applications to be developed as a series of clients that interact remotely with application components and their services through proxies. Jini was chosen as the middleware technology because of its rich support for service-oriented development, but many of the principles apply equally to other middleware technologies, particularly CORBA and Web Services.

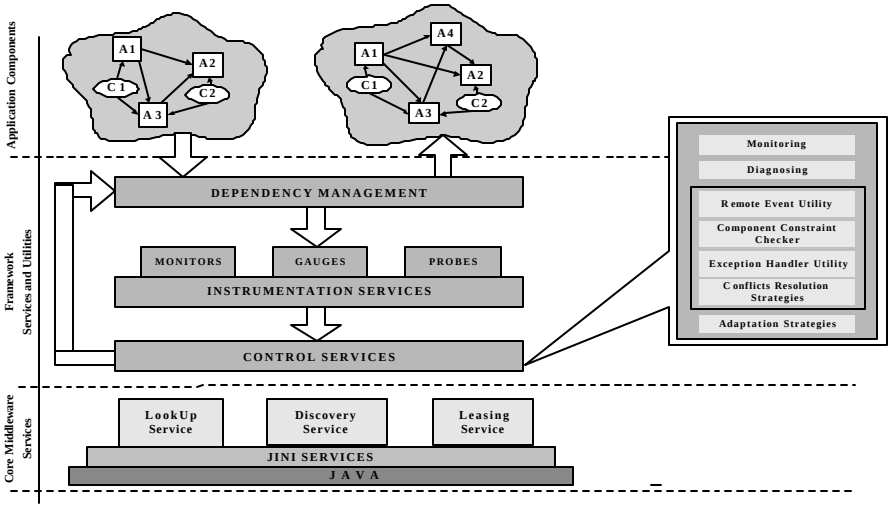


**Figure 2. Framework architecture.**

The framework architecture, shown in Figure 2, is based on a three-layer model: the first layer contains the core Jini middleware services, which include lookup, discovery and leasing. The second layer, at the heart of the framework, contains instrumentation and control services along with dependency management utilities, each of which are explained below:

1. Instrumentation services: monitor and record client access of application services including method invocations. Instrumentation services (considered further in [11]) consist of gauge services, probe services, monitor services and logger and analyzer utilities.
2. Control services: examine information provided by the instrumentation services together with dependency relationships to resolve conflicts through reconfiguration and adaptation. Control services (considered further in [1]) use control rules, activated by conflicts, to identify the cause of a conflict and select a conflict resolution or adaptation strategy. Control services effectively adapt the application through architectural reconfigurations that maintain dependency relationships.
3. Dependency management: maintains the application digraph to provide a faithful "up-to-date" representation of an application's component and service configuration at runtime. Dependency management provides an API that provides access to individual dependency relationships and the dependency digraph as a whole. The API is used by instrumentation and control services to reason about the current state and configuration of an application in order to assess the validity of an adaptation strategy.

The third layer contains the application components and their services, which form the nodes of the dependency graph. As mentioned previously, although the components can be developed independently from the framework services layer they must include a reference to a dedicated manageable proxy. Figure 3 demonstrates how a manageable proxy is attached to a `SimpleService` application component. The reference is provided through a `Manageable` interface that extends Jini's own `Administrable` interface as shown in Figure 3. The `Administrable` interface was included in Jini's API to allow Java objects to be attached to components, at the discretion of the developer, and accessed by a `getAdmin()` method. The manageable proxy, represented as a `ManageableProxy` in Figure 3, implements the `Manageable` interface to provide access to the dependency relationships stored in a `DependencyVector`.

During the development of the framework a design decision was taken to implement the instrumentation services, control services and dependency management utilities as Jini activatable services, which subclass RMI's `Activatable` class. An activatable or "lazy" service, considered further in [14], registers with a Jini lookup service, via a proxy, but after registration, when the activatable service becomes idle, it is allowed to "die" or "sleep", thereby consuming no memory. RMI's daemon, `rmid`, maintains a reference to the dormant service so that it can be resurrected when needed by a client. On first impression, activatable services seem like an attractive option. However, the trade off with activatable services is that although memory usage is reduced a new Java Virtual Machine (JVM) needs to be started when an activatable service is called into use (i.e. its methods are invoked), which can prove detrimental to the performance of an application. However, Jini provides facilities to create activation groups that allow a group of related activatable services to share the same JVM. This facility was used in the framework to create separate activation groups for instrumentation services, control services and dependency management utilities. Furthermore, our previous experiments with the use of activatable services, considered further in [11], justified the design decision to implement instrumentation services, control services and dependency management utilities as activatable services in order to minimize the overhead that they may impose on the application's middleware layer.

```
public class SimpleService extends UnicastRemoteObject {
    protected Manageable proxy;

    public SimpleService() throws RemoteException {
    }

    public Object getManageableProxy() {
        return new ManageableProxy(proxy);
    }
}

public interface Manageable extends Administrable {
    DependencyVector getDependencies();
    ServiceID[] getBindings() throws java.rmi.RemoteException;
    Object[] getTypes();
}
public class ManageableProxy implements Manageable {
    protected Manageable proxy;
    DependencyVector dependencies;

    public ManageableProxy(Manageable proxy) {
        this.proxy = proxy;
    }

    public Object getAdmin() {
        return proxy.getAdmin();
    }

    // further methods to implement getDependencies,
    // getBindings and getTypes
}
```

**Figure 3. Manageable proxy attachment.**

Essentially, the framework operates in a feedback controller regime in that adaptation strategies, to be performed by control services, are checked against the dependency digraph to assess their validity. Adaptations, performed as architectural reconfigurations (i.e. digraph modifications), may in turn cause further conflicts that are fed back, via instrumentation, to the control services. Through this feedback regime the framework behaves as a self-monitoring system for which the performance and behaviour are continually monitored to facilitate the stable operation of the application.

## 5. CASE STUDY

As mentioned previously, the development of the framework is ongoing, but, to date, the basic architecture underlying the framework has already been implemented and tested on an existing Jini application, namely *EmergeITS*[2], [12]. The *EmergeITS* application, shown in Figure 4, is intended to realize the concept of *intelligent networked vehicles*, primarily for use by the emergency fire service. Essentially *EmergeITS* allows emergency fire service personnel to access a variety of application services, from centralized corporate systems, through remote in-vehicle computers and Palm and mobile phone devices. The *EmergeITS* architecture (Figure 4) consists of a collection of service providing components and a Service Manager responsible for registering application components and managing their leases. Services are discovered and used accordingly by one or more in-vehicle client computers.

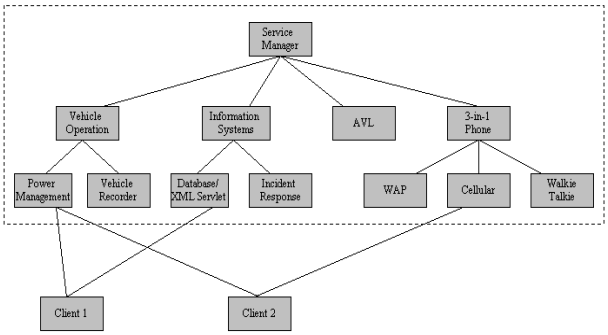


**Figure 4. A simplified view of the EmergeITS architecture.**

For conciseness we only consider the *3-in-1 Phone Service* (right of Figure 4), which allows a mobile phone or Palm device to be used in one of three different modes, subject to the requirements of the user and availability of a communication service provider. For the evaluation trial, monitor instrumentation services and control services were attached, dynamically, to monitor client requests on the 3-in-1 phone service and control the successful use of the service. This dynamic attachment (considered further in [11]) is illustrated in Figure 5, which shows an Instrumentation Factory that produces instrumentation monitor services, on demand, that are attached to EmergeITS application services.

The instrumentation monitor services were used to monitor method invocations made by clients (e.g. `connect()` / `disconnect()` methods) and control service rules were activated as a consequence of any exceptional behaviour. Following such exceptional behaviour, control service rules used the information gathered by monitor services together with dependency relationships, to initiate a conflict resolution strategy to correct the exceptional behaviour. As an example, the current state of the 3-in-1 phone service may indicate a dependency on a WAP gateway and the invocation of the `connect()` method, on the WAP gateway, may result in a `RemoteConnectionException` due to the unavailability of a WAP service.

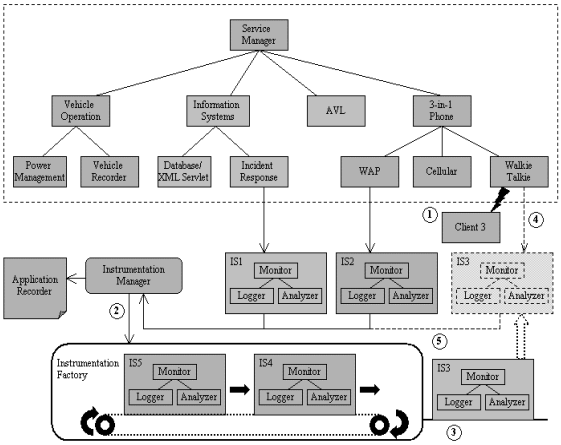


**Figure 5. Instrumentation of 3-in-1 phone service.**

This exception is then checked by a control service rule resulting in the activation of a suitable conflict resolution strategy. The resolution strategy first attempts several connection retries, followed by consecutive pauses. If the retries are unsuccessful, the strategy then searches for an alternative WAP service provider. The latter alternative can be regarded as an adaptation, which is realized as an architectural reconfiguration for which the dependency graph must be updated. Figure 6 shows the specification of the rule that is activated by a `RemoteConnectionException`. The specification of this rule, and others, is based on the *Design by Contract* approach of Meyer, [9], and the rule was implemented as a Jini service using the Java 2 Standard Edition (J2SE).

## 6. CONCLUSIONS

In this paper, we have described a framework, based on the combination of instrumentation and control services and a digraph-based architectural dependency model. We have demonstrated the application of the framework through a case study involving the adaptation of a 3-in-1 phone service to correct runtime conflicts. We feel that the main strength of our approach stems from its ability to simplify adaptation by factoring out the managerial/adaptation code. This alleviates the need to add complex adaptation code to application components and allows instrumentation and control services to concentrate on the manageable proxy objects attached to application components.

In our future work, we intend to extend the portfolio of control services to provide rules to deal with further conflicts including web-server failure, or unavailability, and QoS and security-related conflicts.

[2] *EmergeITS* is a collaborative project between the School of Computing and Mathematical Sciences at Liverpool John Moores University and Merseyside Fire Service <http://www.cms.livjm.ac.uk/emergeits>

```
rule Three_In_1_Connection(Service s,
    LookupList list) : Binding
  tasks
    Connect, Retry, Search
  require
    service_not_null: s not NULL
    lookup_list_not_null: list not NULL
  local
    b : Binding := Binding.FREE
    alternate_s : Service := Service.NULL
  strategy
    do
      b := Retry(s) or
      (alternate_s := Search(s, list)
      ensure
        not_null_service :
          alternate_s not Service.NULL
      then
        b := Connect(alternate_s))
      return b
    end_do
end_rule

task Connect(Service s): Binding
  require
    service_not_null : s not NULL
    max_connections:
     s.num_connections < s.MAX_CONNECTIONS
  local
    b: Binding := Binding.FREE
  do
    b := s.connect()
    return b
  end_do
end_task
```

```
task Retry(Service s) : Binding
  local
    b : Binding := Binding.FREE
    try : INTEGER := 1
  do
    loop
      b := Connect(s)
      pause(10)
      try := try + 1
    until
      (try = 5) or
      (b = Binding.CONNECTION)
    return b
  end_do
end_task

task Search(Service s,
    LookupList list): Service
  local
    temp_l : Lookup := list.firstLookup()
    temp_s : Service :=
                temp_l.firstService()
  do
    ensure
      temp_s.type not s.type
    then
      loop
        loop
          temp_s = temp_l.nextService();
        until
          (temp_s = Lookup.LAST_SERVICE) or
          (temp_s.type = s.type)
        temp_l := list.nextLookup()
      until
        (temp_l = LookupList.LAST_LOOKUP) or
        (temp_s.type = s.type)
    return temp_s
  end_do
end_task
```

**Figure 6. 3-in-1 phone connection rule.**

## 7. REFERENCES


[1] Badr, N., D. Reilly, and A. Taleb-Bendiab, A Conflict Resolution Control Architecture for Self-Adaptive Software, in Proceedings of the International Workshop on Architecting Dependable Systems: WADS 2002 (ICSE 2002), Florida, USA, 2002.

[2] Cheng, S., et al., Using Architectural Style as a basis for System Self-Repair, in The Working IEEE/IFIP Conference on Software Architecture, Montreal, 2002.

[3] Diakov, N.K., et al., Monitoring of Distributed Component Interactions, in Proceedings of RM'2000: Workshop on Reflective Middleware, New York, 2000.

[4] Garlan, D., B. Schmerl, and J. Chang, Using Gauges for Architecture-Based Monitoring and Adaptation, in The Working Conference on Complex and Dynamic Systems Architecture, Brisbane, Australia, 2001.

[5] Hasselmeyer, P., Managing Dynamic Service Dependencies, in 12th International Workshop on Distributed Systems: Operations and Management (DSCOM 2001), France, 2001.

[6] Kokar, M., K. Baslawski, and Y. Eracar, Control Theory-Based Foundation of Self-Controlling Software, IEEE Intelligent Systems, Vol. , No. pp. 37-45, 1999.

[7] Laddaga, R., Active Software, in 1st International Workshop on Self-Adaptive Software (IWSAS2000), Oxford, UK, Springer-Verlag,2000.

[8] Meng, A.C., On the Evaluation of Self-Adaptive Software, in 1st International Workshop On Self-Adaptive Software (IWSAS2000), Oxford, UK, Springer-Verlag,2000.

[9] Meyer, B., Object-Oriented Software Construction, 2nd ed, Prentice Hall, New Jersey, 1997.

[10] Osterweil, L.J. and L.A. Clarke, Continuous Self-Evaluation for the Self-Improvement of Software, in 1st International Workshop on Self-Adaptive Software (IWSAS2000), Oxford, UK, Springer-Verlag,2000.

[11] Reilly, D. and A. Taleb-Bendiab, Dynamic Software Instrumentation for Jini Applications, in Proceedings of the 3rd International Workshop on Software Engineering Middleware SEM 2002 (ICSE 2002), Orlando, Florida, USA, Springer-Verlag,2002.

[12] Reilly, D. and A. Taleb-Bendiab, A Service-Based Architecture for In-Vehicle Telematics Systems, in IEEE Proceedings of the 22nd International Conference on Distributed Computing Systems (ICDCS 2002) WORKSHOPS - International Workshop of Smart Appliance and Wearable Computing (IWSAWC 2002), Vienna, Austria, 2002.

[13] Robertson, P., R. Laddaga, and H. Shrobe, Introduction to the 1st International Workshop on Self-Adaptive Software, in 1st International Workshop on Self-Adapative Software, Oxford, UK, Springer-Verlag, 2000.

[14] Sun Microsystems Inc., Jini Architecture Specification - v1.1, http://www.sun.com/jini/specs, 2000.