A Lightweight Infrastructure for Reconfiguring Applications

Marco Castaldi[1], Antonio Carzaniga[2], Paola Inverardi[1], and Alexander L. Wolf[2]

[1] Dipartimento di Informatica
Universita' dell'Aquila
via Vetoio, 1 67100 L'Aquila, Italy
{castaldi,inverard}@di.univaq.it
[2] Department of Computer Science
University of Colorado at Boulder
Boulder, Colorado, 80309-0430 USA
{carzanig,alw}@cs.colorado.edu

Abstract. We describe Lira, a lightweight infrastructure for managing dynamic reconfiguration that applies and extends the concepts of network management to component-based, distributed software systems. Lira is designed to perform both component-level reconfigurations and scalable application-level reconfigurations, the former through agents associated with individual components and the latter through a hierarchy of managers. Agents are programmed on a component-by-component basis to respond to reconfiguration requests appropriate for that component. Managers embody the logic for monitoring the state of one or more components, and for determining when and how to execute reconfiguration activities. A simple protocol based on SNMP is used for communication among managers and agents.

1 Introduction

This paper addresses the problem of managing the dynamic reconfiguration of component-based, distributed software systems. Reconfiguration comes in many forms, but two extreme approaches can be identified: *internal* and *external*.

Internal reconfiguration relies on the programmer to build into a component the facilities for reconfiguring the component. For example, a component might observe its own performance and switch from one algorithm or data structure to another when some performance threshold has been crossed. This form of reconfiguration is therefore sometimes called "programmed" or "self-healing" reconfiguration.

External reconfiguration, by contrast, relies on some entity external to the component to determine when and how the component is reconfigured. For example, an external entity might monitor the performance of a component and perform a wholesale replacement of the component when a performance threshold has been crossed. Likewise, an external entity might determine when to

B. Westfechtel, A. van der Hoek (Eds.): SCM 2001/2003, LNCS 2649, pp. 231–244, 2003.

upgrade a component to a newer version and perform that upgrade by replacing the component within the application system. Of course, replacing a component is a rather drastic reconfiguration action, but there is little that an external entity can do without cooperation from the component itself.

One common form of cooperation is the provision by a component of reconfiguration parameters; the parameters define what internal reconfigurations a component is prepared to carry out, while an external entity is given the ability to set parameter values and thereby to determine which of the possible reconfigurations is to occur and when. Clearly, any particular approach to reconfiguration is likely to be some blending of the two extreme approaches in conjunction with the use of reconfiguration parameters.

At a level above the individual components, we can consider the reconfiguration of the larger application system, where the dominant concern is the topology of the application in terms of the number and location of its components. What this typically introduces into the problem is the need to carry out a coordinated set of reconfigurations against the individual components. The question then arises, how and where is this coordination activity specified and managed?

The "easy" answer would be some centralized, external entity. However, the viability of such an entity essentially assumes that (a) components are designed and built to cooperate with the external entity and (b) it is possible for the entity to have global knowledge of the state of the application. These assumptions run counter to modern development methodology: we want to build generic components having few dependencies so that they can be reused in multiple contexts, and we want distributed systems to be built without global knowledge so that they can scale and be resilient to failure.

In previous work, we developed the Software Dock software deployment system [11]. The Software Dock is a comprehensive tool that addresses issues such as configuration, installation, and automated update. It also explores the challenges of representing component dependencies and constraints arising from heterogeneous deployment environments [10]. The Software Dock provides an extensive and sophisticated infrastructure in which to define and execute post-development activities [12]. However, it does not provide explicit support for dynamic reconfiguration—that is, a reconfiguration applied to a running system—although its infrastructure was designed to accommodate the future introduction of such a capability.

In an effort to better understand the issues surrounding dynamic reconfiguration, we developed a tool called Bark [21]. In contrast to the Software Dock, which is intended to be generic, Bark is a reconfiguration tool that is designed specifically to work within the context of the EJB (Enterprise JavaBean) [20] component framework. Its infrastructure leverages and extends the EJB suite of services and is therefore well integrated into a standard platform. Of course, its strength is also its weakness, since this tight integration means that it is useful only to application systems built on the EJB model. On the other hand, we learned an important lesson from our experience with Bark, namely that it is both possible and advantageous to make use of whatever native facilities are al-

ready provided by the components for the purposes of dynamic reconfiguration. Moreover, the burden of tailoring reconfiguration activities is naturally left to and divided among the developers of individual components, the developers of subsystems of components, and ultimately the developers of the encompassing applications.

Reflecting back, then, on our experience with the Software Dock, we realized that it imposes rather severe demands on component and application developers, above and beyond any necessary tailoring. In particular, the architecture of the Software Dock requires that at least one so-called *field dock* reside on every host machine. The field dock serves as the execution environment for all deployment activities, the store for all data associated with deployment activities, and the interface to the file system on the host. The field dock is also the mediator for all communication between individual components and external entities having to do with deployment activities. Finally, in order to make use of the Software Dock, developers must encode detailed information about their components and applications in a special deployment language called the Deployable Software Description (DSD).

Thus, the Software Dock would lead to what we now consider a "heavy-weight" solution to the problem of dynamic reconfiguration, a characteristic shared by many other reconfiguration systems (e.g., DRS [1], Lua [2], and PRISMA [3]). While such an approach may be feasible in some circumstances, we are intrigued by the question of how to build lighter-weight solutions.

It is difficult to be precise about what one means by "lightweight", since it is inherently a relative concept. But for our purposes, we take lightweight to indicate intuitively an approach to dynamic reconfiguration in which:

- the service is *best effort*, in that it arises from, and makes use of, the facilities already provided by individual components, rather than some standardized set of imposed facilities;
- reconfiguration is carried out via *remote control*, in that the management of reconfiguration is separated from the implementation of reconfiguration, so as to enhance the reusability of components and, in conjunction with the best-effort nature of the service, broaden the scope of applicability; and
- communication is through a *simple protocol* between components and the entities managing their reconfiguration, rather than through complex interfaces and/or data models.

In this paper we describe our attempt at a lightweight infrastructure for dynamic reconfiguration and its implementation in a tool called Lira. The inspiration for our approach comes directly from the field of network management and its Internet-Standard Network Management Framework, which for historical reasons is referred to as SNMP [6]. (SNMP is the name of the protocol used within the framework.) Our hypothesis is that this framework can serve, with appropriate extension and adaptation where necessary, as a useful model for lightweight reconfiguration of component-based, distributed software systems.

In the next section, we provide background on network management and the Internet-Standard Network Management Framework. Section 3 describes

Lira, while Section 4 presents a brief example application of Lira that we have implemented. Related work is discussed in Section 5, and we conclude in Section 6 with a discussion of future work.

2 Background: Network Management

As mentioned above, the design of Lira was inspired by network management approaches. The original challenge for network management was to devise a simple and lightweight method for managing network devices, such as routers and printers. These goals were necessary in order to convince manufacturers to make their devices remotely manageable without suffering undue overhead, as well as to encourage widespread acceptance of a method that could lead to a *de facto* management standard. (Perhaps the same can be said of software component manufacturers.)

The network management model consists of four basic elements: *agents*,[1] each of which is associated with a network node (i.e., device) to be managed and which provides remote management access on behalf of the node; *managers*, which embody the logic for monitoring the state of a node and for determining when and how to execute management activities; *a protocol*, which is used for communication among the agent and manager management entities; and *management information*, which defines the aspects of a node's state that can be monitored and the ways in which that state can be modified from outside the node.

Agents are typically provided by node manufacturers, while managers are typically sophisticated third-party applications (e.g., HP's OpenView [14]). The standard protocol is SNMP (Simple Network Management Protocol), which provides managers with the primitive operations for getting and setting variables on agents, and for sending asynchronous alerts from agents to managers. The management information defines the state variables of an agent and is therefore specific to each node. These variable definitions are captured in a MIB (Management Information Base) associated with each agent.

In our work on Lira, we are driven by the complexity of configurations inherent in today's large-scale, component-based, distributed software systems. Specifically, multiple components tend to execute on the same device, and regularly come into and go out of existence (much more often than, say, a router in a network). Further, the components, whether executing on the same or on different devices, tend to have complex relationships and interdependencies. Finally, domains of authority over components tend to overlap and interact, implying complex management relationships.

In theory, the Internet-Standard Network Management Framework places few constraints on how its simple concepts are to be applied, allowing for quite

[1] The term "agent" as used in network management should not be confused with other uses of this term in computer science, such as "mobile agent" or "intelligent agent". Network management agents are not mobile, and their intelligence is debatable.

advanced and sophisticated arrangements. In practice, however, network management seems to have employed these concepts in only relatively straightforward ways. For instance, a typical configuration for managing a network consists of a flat space of devices with their associated agents managed by a single, centralized manager; a manager is associated with a particular domain of authority (e.g., a business organization) and controls all the devices within that domain. It is interesting to note that there have been efforts at defining MIBs for some of the more prominent web applications, such as the Apache web server, IBM's WebSphere transaction server, and BEA's WebLogic transaction server, and more generally a proposal for an "application management" MIB [15]. But, again, the approach taken is to view these as independently managed applications, not a true complex of distributed components.

3 Lira

The essence of the approach we take in Lira is to define a particular method for applying the basic facilities of the Internet-Standard Network Management Framework to complex component-based software systems. To summarize:

- We distinguish two kinds of agent. A *reconfiguration agent* is associated with a component, and is responsible for reconfiguring the component in response to operations on variables defined by its MIB. A *host agent* is associated with a computer in the network, and is responsible for installing and activating components on that computer, again, in response to operations on variables defined by its MIB.
- A manager can itself be a reconfiguration agent. What this means is that a manager can have a MIB and thereby be expected to respond to other, higher-level managers. Such a *manager agent* would reinterpret the reconfiguration (and status) requests it receives into management requests it should send to the agents of the components it is managing. In this way a scalable management hierarchy can be established, finally reaching ground on the *base* reconfiguration agents associated with (monolithic) components.
- We define a basic set of "standard" MIB definitions for each kind of agent. These definitions are generically appropriate for managing software components, but are expected to be augmented on an agent-by-agent basis so that individual agents can be specialized to their particular unique tasks.

It is important to note that Lira does not itself provide the agents, although in our prototype implementation we have created convenient base classes from which implementations can be derived. Rather, the principle that we follow is that developers should be free to create agents in any programming language using any technology they desire, as long as the agents serve their intended purpose and provide access through at least the set of MIB definitions we have defined. For example, in our use of Lira within the Willow survivability middleware system [16], there are agents written in C++ and Java. Furthermore, some

of the Willow manager agents are highly sophisticated workflow engines that can coordinate and adjudicate among competing reconfiguration requests [17].

The remainder of this section describes these concepts in greater detail.

Reconfiguration Agent

A base reconfiguration agent directly controls and manages a component. Lira does not constrain how the agent is associated with its component, only that the agent is able to act upon the component. For example, the agent might be part of the same thread of execution, execute in a separate thread, or execute in a completely separate process. In fact, the agent might reside on a completely different device, although this would probably be the case only for complex agents associated with components running on capacity-limited devices.

The logical model of communication between a base reconfiguration agent and its component is through shared memory; the component shares a part of its state with the agent. Of course, to avoid synchronization problems, the component must provide atomic access to the shared state.

A reconfiguration agent that is not a base reconfiguration agent is a manager. It interacts with other base and non-base reconfiguration agents using the standard management protocol. For purposes of simplifying the discussion below, we abuse the term "component" to refer also to the subassembly of agents with which a non-base reconfiguration agent (i.e., a manager agent) interacts. Thus, from the perspective of a higher-level manager, it appears as though a lower-level manager is any other reconfiguration agent. This is illustrated in the example agent hierarchy of Figure 1.

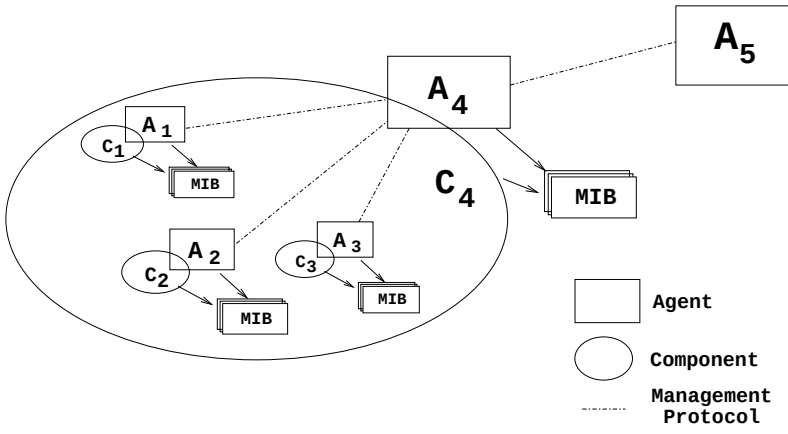

Fig. 1. An example Lira agent hierarchy

In the figure, agents A_1, A_2, and A_3 are base reconfiguration agents acting on components C_1, C_2, and C_3, respectively. Agent A_4 is a manager for A_1, A_2,

and A_3, but is treated as a reconfiguration agent by the higher-level manager A_5. In effect, A_4 is responsible for carrying out reconfigurations on a subsystem represented by C_4, and hides the complexity of that responsibility from A_5.

A reconfiguration agent is essentially responsible for managing the lifecycle of its component, and exports at least the following five management functions:[2]

- `void START(startArgs)`
- `void STOP()`
- `void SUSPEND()`
- `void RESUME()`
- `void SHUTDOWN()`

The function `SHUTDOWN` also serves to terminate the agent. Each reconfiguration agent also exports at least the following two variables:

- `STATUS`
- `NOTIFYTO`

The first variable contains the current status of the component, and can take on one of the following values: *starting*, *started*, *stopping*, *stopped*, *suspending*, *suspended*, and *resuming*. The second variable contains the address of the manager to which an alert notification should be sent. This is necessary because we allow agents to have multiple managers, but we assume that at any given time only one of those managers has responsibility for alerts. That manager can use other means, not defined by Lira, to spread the alert.

Host Agent

A host agent runs on a computer where components and reconfiguration agents are to be installed and activated, and is responsible for carrying out those activities in response to requests from a manager. (How a host agent is itself installed and activated is obviously a bootstrapping process.) As part of activating a component and its associated agent, the host agent provides an available network port, called the *agent address*, to the reconfiguration agent over which that agent can receive requests from a manager.

Each host agent exports at least the following variables:

- `NOTIFYTO`
- `INSTALLEDAGENTS`
- `ACTIVEAGENTS`

Host agents also export the following functions:

- `void INSTALL(componentPackage)`
- `void UNINSTALL(componentPackage)`
- `agentAddress GET_AGENTADDRESS(agentName)`

[2] Lira provides the notion of a "function", described below, as a convenient shorthand for a combination of more primitive concepts already present in SNMP.

- `agentAddress ACTIVATE(componentType,componentName,componentArgs)`
- `void DEACTIVATE(componentName)`
- `void REMOVEACTIVEAGENT(agentName)`

Again, we expect host agents to export additional variables and functions consistent with their particular purposes, including variables described in the Host Resources MIB [24].

Management Protocol

The management protocol follows the SNMP paradigm. Each message in the protocol is either a *request* or a *response*, as shown in the following table:

request	*response*
SET(*variable_name, variable_value*)	ACK(*message_text*)
GET(*variable_name*)	REPLY(*variable_name, variable_value*)
CALL(*function_name, parameters_list*)	RETURN(*return_value*)

Requests are sent by managers to agents, and responses are sent back to managers from agents. There is one additional kind of message, which is sent from agents to managers in the absence of any request.

NOTIFY(*variable_name, variable_value, agent_name*)

This message is used to communicate an alert from an agent back to a manager. For instance, an agent might notice that a performance threshold has been crossed, and uses the alert to initiate some remedial action on the part of the manager.

4 Example

We now present a simple, yet practical example to demonstrate how Lira can be used to achieve a dynamic reconfiguration. The example was implemented using Java agents working on components of a pre-existing Java application. Lira has been used in more complex and diverse settings, but this example suffices for illustrative purposes.

The application is an overlay network of software routers for a distributed, content-based, publish/subscribe event notification service called Siena [5]. The routers form a store-and-forward network responsible for delivering messages posted by publishers on one side of a network to the subscribes having expressed interest in the message on the other side of the network. Publishers and subscribers are clients of the service that can connect to arbitrary routers in the network. The routers are arranged in a hierarchical fashion, such that each has a unique parent, called a *master*, to which subscription and notification messages are forwarded. (Notification messages also flow down the hierarchy, from parents to children, but that fact is not germane to this example.) The master of a client

is the router to which it is attached. Siena is designed to adjust its forwarding tables in response to changes in subscriptions, and also in response to changes in topology. The topology can be changed through a Siena command called *set master*, which resets the master of a given router.

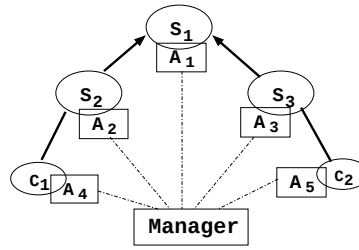

Fig. 2. Topology of an example Siena network

Figure 2 shows a simple topology, where S_1, S_2, and S_3 are Siena routers, and C_1 and C_2 are Siena clients. S_1 is the master of both S_2 and S_3. Each router and client has associated with it a reconfiguration agent. All the agents are managed by a single manager. In addition to the "standard" set of variables, each reconfiguration agent in this system exports a variable `MASTER` to indicate the identity of its component's master router.

The Siena clients and routers, together with their reconfiguration agents, can each be run on separate computers. Each such computer would have its own host agent. The manager interacts with the host agent to activate a client or router. For example, the manager uses the function

```
ACTIVATE(SienaRouter,S1,
           "-host palio.cs.colorado.edu -port 3333 -log -")
```

to activate Siena router S_1.

Now, notice that if S_1 were to fail, then clients C_1 and C_2 would not be able to communicate. In such a case, we would like to reconfigure the Siena network to restore communication. The manager can do this by reassigning S_3 to be the master of S_2, as shown in Figure 3. The manager will change the value of the variable `MASTER` of agents A_2 and A_3, sending the request `SET("MASTER",`S_3`)` to A_2 and the request `SET("MASTER","")` to A_3.

Clearly, for the manager to be able to decide on the proper course of action, the state of the Siena network must be monitored. Moreover, the manager must have knowledge of the current topology. This can be done in several ways using Lira, including requests for the values of appropriate variables and the use of the `NOTIFY` message when an agent notices that a router is unresponsive.

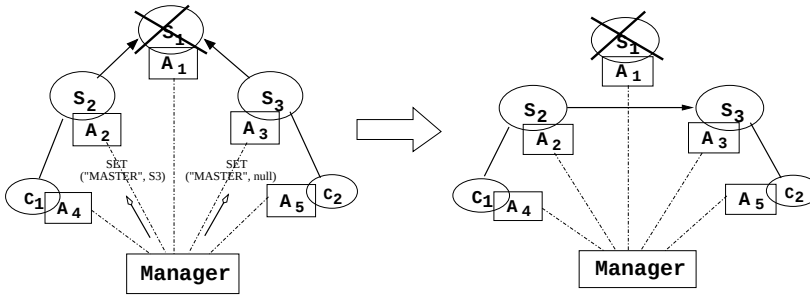

Fig. 3. Reconfiguration in response to the failure of S_1

5 Related Work

Supporting the dynamic reconfiguration of distributed systems has been a goal of researchers and practitioners for the past quarter century, and many techniques and tools have been developed. The work described in this paper is leveraging and integrating the recent maturation of two disciplines, component-based software engineering [13] and network management [6].

Endler divides dynamic reconfiguration into two forms according to when the change is defined: *programmed* and *ad hoc* [8]. The first form is defined at the time the system is designed, and may be compiled into the code of the application. The second form is not predictable and defined only once the application is already in execution. To a certain extent, Lira supports the use of both forms of reconfiguration, the first through planned requests directed at reconfiguration agents, and the second either through replacement of components or through topological reconfigurations at the application level. Of course, the goal of Lira is to automate reconfiguration activities, so the reconfigurations cannot be completely unplanned unless control is given over to the ultimate manager, the human operator.

Endler also discusses a distinction between *functional* and *structural* dynamic reconfigurations [8]. Functional reconfiguration involves new code being added to an application, while structural reconfiguration is topological in nature. Again, Lira supports both.

Several researchers, including Almeida et al. [1], Bidan et al. [4], Kramer and Magee [18], and Wermelinger [25], have tried to address the problem of maintaining consistency during and after a reconfiguration. Usually, the consistency properties of the system are expressed through logical constraints that should be respected, either *a posteriori* or *a priori*. If the constraints are seen as post-conditions [26], the reconfiguration must be undone if a constraint is violated. If the constraints are seen as preconditions [8], the reconfiguration can be done only if the constraints are satisfied.

Lira approaches the consistency problem using a sort of "management by delegation" [9], in which it delegates responsibility to agents to do what is necessary to guarantee consistency and state integrity. This is in line with the idea

of "self-organizing software architectures" [19], where the goal is to minimize the amount of explicit management and reduce protocol communication. It is also in line with the idea of a lightweight, best-effort service (see Section 1), where we assume that the component developer has the proper insight about how best to maintain consistency. This is in contrast to having the reconfiguration infrastructure impose some sort of consistency-maintenance scheme of its own.

Java Management eXtensions (JMX) is a specification that defines an architecture, design pattern, APIs, and services for application and network management in the Java programming language [23]. Under JMX, each managed resource and its reconfiguration services are captured as a so-called MBean. The MBean is registered with an MBean server inside a JMX agent. The JMX agent controls the registered resources and makes them available to remote management applications. The reconfiguration logic of JMX agents can be dynamically extended by registering MBeans. Finally, the JMX specification allows communication among different kinds of managers through connectors and protocol adaptors that provide integration with HTTP, RMI, and even the SNMP protocols. While JMX is clearly a powerful reconfiguration framework, it is also a heavyweight mechanism, and one that is strongly tied to one specific platform, namely Java.

6 Conclusions and Future Work

Lira represents our attempt to devise a lightweight infrastructure for the dynamic reconfiguration of component-based, distributed software systems. Lira follows the approach pioneered in the realm of network management, providing in essence a particular method for applying the concepts of the Internet-Standard Network Management Framework. Lira is designed to perform both component-level reconfigurations and scalable application-level reconfigurations, the former through agents associated with individual components and the latter through a hierarchy of managers.

A hierachical approach is neither new nor necessary, but it seems to us to be natural and suitable for the purpose, allowing the reconfiguration developer to concentrate on the logic of the reconfiguration rather than on how to coordinate the agents. We are currently investigating more sophisticated approaches to supporting cooperation among agents and, consequently, for making more complex coordinated reconfiguration decisions (see below).

Lira has been developed with the aim of providing a minimal set of functionality. Some desirable capabilities found in more "heavyweight" approaches, such as automated consistency management and version selection, are not provided as part of the basic infrastructure. However, two main strategies can be followed to bridge the gap: (1) advanced capabilities can be used from, or implemented within, individual agents, thus hiding them from managers (i.e., from higher-level agents) or (2) they can be used or implemented at the management level, exploiting variables and functions provided by lower-level agents. It is still

an open question as to whether this lightweight approach and its flexible programmable extension is superior to a heavyweight approach and its implicit set of capabilities. We are trying to answer this question through case studies.

We have implemented a prototype of the Lira infrastructure and used it to manage several complex distributed applications, including a network of Siena overlay routers [5] and a prototype of a military information fusion and dissemination system called the Joint Battlespace Infosphere [22]. Based on these and other experiences, we have begun to explore how the basic Lira infrastructure could be enhanced in certain specialized ways.

First, in order to simplify the exportation of reconfiguration variables and functions for Java components, we have created a specialized version of the Lira agent. This agent uses the Java Reflection API to provide mechanisms to export (public) variables and functions defined in the agent and/or in the component. These exported entities are then integrated into the MIB for the agent using a callback mechanism.

Second, in order to provide more "intelligence" in reconfiguration agents, we have created an API to integrate Lira with a ProLog-like language called DALI [7]. The idea is to be able to implement agents that can reason about the local context, make decisions based on that reasoning, and remember (or learn) from past situations. The result of this integration is an intelligent agent we call LiDA (Lira + DALI), which is more autonomous than a basic Lira agent and uses its intelligence and memory to make some simple, local decisions without support from a manager.

Finally, we have created a preliminary version of a reconfiguration language that allows one to define, in a declarative way, application-level reconfiguration activities. We have observed that Lira agents operating at this level follow a regular structure in which their concern is centered on the installation/activation of new components, changes in application topology, and monitoring of global properties of the system. The particulars of these actions can be distilled out and used to drive a generic agent. This echos the approach we took in the Software Dock, where generic agents operate by interpreting the declarative language of the DSD [11].

None of these enhancements are strictly necessary, but they allow us to better understand how well Lira can support sophisticated, programmer-oriented specializations, which is a property that we feel will make Lira a broadly acceptable, lightweight reconfiguration framework.

Acknowledgements

The authors thank Dennis Heimbigner, Jonathan Hill, and John Knight for their helpful comments, criticism, and feedback on the design and prototype implementation of Lira.

This work was supported in part by the Defense Advanced Research Projects Agency, Air Force Research Laboratory, Space and Naval Warfare System Center, and Army Research Office under agreement numbers F30602-01-1-0503,

F30602-00-2-0608, N66001-00-1-8945, and DAAD19-01-1-0484. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Defense Advanced Research Projects Agency, Air Force Research Laboratory, Space and Naval Warfare System Center, Army Research Office, or the U.S. Government.

References

1. J.P.A. Almeida, M. Wegdam, M. van Sinderen, and L. Nieuwenhuis. Transparent Dynamic Reconfiguration for CORBA. In *Proceedings of the 3rd International Symposium on Distributed Objects and Applications*, pages 197–207. IEEE Computer Society, September 2001.
2. T. Batista and N. Rodriguez. Dynamic Reconfiguration of Component-Based Applications. In *Proceedings of the International Symposium on Software Engineering for Parallel and Distributed Systems*, pages 32–39. IEEE Computer Society, June 2000.
3. J. Berghoff, O. Drobnik, A. Lingnau, and C. Monch. Agent-Based Configuration Management of Distributed Applications. In *Proceedings of the 3rd International Conference on Configurable Distributed Systems*, pages 52–59. IEEE Computer Society, May 1996.
4. C. Bidan, V. Issarny, T. Saridakis, and A. Zarras. A Dynamic Reconfiguration Service for CORBA. In *Proceedings of the 4th International Conference on Configurable Distributed Systems*, pages 35–42. IEEE Computer Society, May 1998.
5. A. Carzaniga, D.S. Rosenblum, and A.L. Wolf. Design and Evaluation of a Wide-Area Event Notification Service. *ACM Transactions on Computer Systems*, 19(3):332–383, August 2001.
6. J. Case, R. Mundy, D. Partain, and B. Stewart. *Introduction and Applicability Statements for Internet Standard Management Framework*. RFC 3410. The Internet Society, December 2002.
7. S. Costantini and A. Tocchio. A Logic Programming Language for Multi-Agent Systems. In *8th European Conference on Logics in Artificial Intelligence*, number 2424 in Lecture Notes in Artificial Intelligence, pages 1–13. Springer-Verlag, September 2002.
8. M. Endler. A Language for Implementing Generic Dynamic Reconfigurations of Distributed Programs. In *Proceedings of 12th Brazilian Symposium on Computer Networks*, pages 175–187, 1994.
9. G. Goldszmidt and Y. Yemini. Distributed Management by Delegation. In *Proceedings of the 15th International Conference on Distributed Computing Systems*, pages 333–340. IEEE Computer Society, May 1995.
10. R.S. Hall, D.M. Heimbigner, and A.L. Wolf. Evaluating Software Deployment Languages and Schema. In *Proceedings of the 1998 International Conference on Software Maintenance*, pages 177–185. IEEE Computer Society, November 1998.
11. R.S. Hall, D.M. Heimbigner, and A.L. Wolf. A Cooperative Approach to Support Software Deployment Using the Software Dock. In *Proceedings of the 1999 International Conference on Software Engineering*, pages 174–183. Association for Computer Machinery, May 1999.

12. D.M. Heimbigner and A.L. Wolf. Post-Deployment Configuration Management. In *Proceedings of the Sixth International Workshop on Software Configuration Management*, number 1167 in Lecture Notes in Computer Science, pages 272–276. Springer-Verlag, 1996.
13. G.T. Heineman and W.T. Councill, editors. *Component-Based Software Engineering: Putting the Pieces Together*. Addison-Wesley, Reading, Massachusetts, 2001.
14. Hewlett Packard. *HP OpenView Family Guide*, 1998.
15. C. Kalbfleisch, C. Krupczak, R. Presuhn, and J. Saperia. *Application Management MIB*. RFC 2564. The Internet Society, May 1999.
16. J.C. Knight, D.M. Heimbigner, A.L. Wolf, A. Carzaniga, J. Hill, and P. Devanbu. The Willow Survivability Architecture. In *Proceedings of the Fourth International Survivability Workshop*, March 2002.
17. J.C. Knight, D.M. Heimbigner, A.L. Wolf, A. Carzaniga, J. Hill, P. Devanbu, and M. Gertz. The Willow Architecture: Comprehensive Survivability for Large-Scale Distributed Applications. Technical Report CU-CS-926-01, Department of Computer Science, University of Colorado, Boulder, Colorado, December 2001.
18. J. Kramer and J. Magee. Dynamic Configuration for Distributed Systems. *IEEE Transactions on Software Engineering*, SE-11(4):424–436, April 1985.
19. J. Magee and J. Kramer. Self Organising Software Architectures. In *Proceedings of the Second International Software Architecture Workshop*, pages 35–38, October 1996.
20. R. Monson-Haefel. *Enterprise JavaBeans*. O'Reilly and Associates, 2000.
21. M.J. Rutherford, K.M. Anderson, A. Carzaniga, D.M. Heimbigner, and A.L. Wolf. Reconfiguration in the Enterprise JavaBean Component Model. In *Proceedings of the IFIP/ACM Working Conference on Component Deployment*, number 2370 in Lecture Notes in Computer Science, pages 67–81. Springer-Verlag, 2002.
22. Scientific Advisory Board. Building The Joint Battlespace Infosphere. Technical Report SAB-TR-99-02, U.S. Air Force, December 2000.
23. Sun Microsystems, Inc., Palo Alto, California. *Java Management Extensions Instrumentation and Agent Specification, v1.0*, July 2000.
24. S. Waldbusser and P. Grillo. *Host Resources MIB*. RFC 2790. The Internet Society, March 2000.
25. M. Wermelinger. Towards a Chemical Model for Software Architecture Reconfiguration. In *Proceedings of the 4th International Conference on Configurable Distributed Systems*, pages 111–118. IEEE Computer Society, May 1998.
26. A. Young and J. Magee. A Flexible Approach to Evolution of Reconfigurable Systems. In *Proceedings of the IEE/IFIP International Workshop on Configurable Distributed Systems*, pages 152–163, March 1992.