A Framework for Reconfiguration-based Fault-Tolerance in Distributed Systems

Stefano Porcarelli[1], Marco Castaldi[2], Felicita Di Giandomenico[1], Andrea Bondavalli[3], and Paola Inverardi[2]

[1] Italian National Research Council, ISTI Dept., via Moruzzi 1, I-56124, Italy
{porcarelli, digiandomenico}@isti.cnr.it
[2] University of L'Aquila, Dip. Informatica, via Vetoio 1, I-67100, Italy
{castaldi, inverard}@di.univaq.it
[3] University of Firenze, Dip. Sistemi e Informatica, via Lombroso 67/A, I-50134, Italy
a.bondavalli@dsi.unifi.it

Abstract. Nowadays, many critical services are provided by complex distributed systems which are the result of the reuse and integration of a large number of components. Given their multi-context nature, these components are, in general, not designed to achieve high dependability by themselves, thus their behavior with respect to faults can be the most disparate. Nevertheless, it is paramount for these kinds of systems to be able to survive failures of individual components, as well as attacks and intrusions, although with degraded functionalities. To provide control capabilities over unanticipated events, we focus on fault handling strategies, particularly on system's reconfiguration. The paper describes a framework which provides fault tolerance of components based applications by detecting failures through monitoring and by recovering through system reconfiguration. The framework is based on Lira, an agent distributed infrastructure for remote control and reconfiguration, and a decision maker for selecting suitable new configurations. Lira allows for monitoring and reconfiguration at components and applications level, while decisions are taken following the feedbacks provided by the evaluation of statistical Petri net models.

1 Introduction

Dependability is becoming a crucial requirement of current computer and information systems, and it is foreseeable that its importance will increase in the future at a fast pace. We are witnessing the construction of complex distributed systems, for example in the telecommunication or financial domain, which are the result of the integration of a large number of low–cost, relatively unstable COTS (Commercial Off–The–Shelf) components, as well as of previously isolated legacy systems. The resulting systems are being used to provide services which have become critical in our everyday life. Since COTS and legacy components are not designed to achieve high dependability by themselves, their behavior with respect to faults can be the most disparate. Thus, it is paramount for these

kinds of systems to be able to survive failures of individual components, as well as attacks and intrusions, although with degraded functionalities.

The management of non functional properties in heterogeneous, complex component-based systems raises many issues to the application developers. Firstly, the management of such properties is difficult when the source code of the components is not available (*black box components*), since the developer has the limitation of using only the component's public API. Secondly, the heterogeneous environment, where components often implement different error detection and recovery techniques, without any or poor coordination among them, makes the composition of the non-functional properties a very difficult task. Moreover, a distributed application is exposed to communication and coordination failures, as well as to hardware or operating systems ones, that cannot be managed at component level but only at application level.

In order to provide fault tolerance of distributed component based applications, the system developer is forced to implement mechanisms which take into account the fault tolerance policies implemented by the different components within the system, and add the necessary coordination support for the management of fault tolerance at application level. Even if the creation of *ad hoc* solutions is possible and still very popular, some innovative approaches to face this problem have been recently proposed [1, 2].

In this paper we present a methodology and a framework for fault tolerance provision in distributed applications, created by assembling COTS and legacy components together with *ad hoc* application dependent components. The approach is based on monitoring the managed application to detect failures at component and operating system level, and on using dynamic reconfiguration for error recovery and/or for maintaining the system in a certain desirable state.

How to reconfigure the system is decided at run time, following a set of pre-specified reconfiguration policies. The decision process is performed by using online evaluation of a stochastic dependability model which represents the whole system. Such modeling activity depends on the specified policy, on the requirements of the application and on the system status at reconfiguration time. In order to represent the topology of the managed application, which may change dynamically, the model is created at run time by assembling a set of sub–models (building blocks). Before the evaluation, these sub–models are opportunely instantiated to represent the single parts of the system (such as components, hosts and connectors) and initialized with information collected at run time. When multiple reconfigurations are eligible, it seems reasonable in the context of fault tolerance provision to prefer the one that minimizes the error proneness of the resulting system.

The effectiveness of a reconfiguration policy depends on an accurate diagnosis of the nature of the unanticipated event, namely whether a hard, physical fault is affecting the system, or environmental adverse conditions are causing a soft fault which will naturally disappear in some time. However, it is out of the scope of this paper to address diagnosis issues, and we concentrate on reconfiguration only. Our proposed framework for fault tolerance provision is based on *Lira*, an

infrastructure which monitors the system status and implements the reconfiguration strategy, enriched with a model based *Decision Maker*, which decides the most rewarding new reconfiguration for the managed application.

The paper is organized as follows. Section 2 introduces the reconfiguration based approach used for fault tolerance provision, while Section 3 describes the proposed framework. Sections 3.1 and 3.3 detail the different parts of the framework, with particular attention to the Lira reconfiguration infrastructure and the model based decision making process. Section 4 presents an illustrative example to show how the framework works, and Section 5 overviews related work. Finally, Section 6 summarizes the contribution of the paper.

2 A reconfiguration based approach to fault tolerance

In the proposed approach the dynamic reconfiguration plays a central role: the managed system is reconfigured as a reaction to a specified event (for example, a detected failure) or as an optimization operation. A *reconfiguration* can be defined as "any change in the configuration of a software application" [3, 4]. Depending on where these changes occur, different kinds of reconfiguration are defined: a *component level reconfiguration* is defined as any change in the configuration parameters of single components, also called component re–parameterization, while an *architectural level reconfiguration* is any change in the application topology in terms of number and locations of software components [5–8].

The actuation of both kinds of reconfigurations should be performed in a coordinated way: if the components are self-contained and loosely coupled, a component level reconfiguration usually does not impact other components within the system; on the contrary, if there are many dependencies among components, or in the case of architectural reconfigurations which involve several components, a coordinator is necessary for a correct actuation of the reconfiguration process. The coordinator is in charge of considering both dependencies among components and architectural constraints in terms of application topology, in order to maintain the consistency within the system during and after the reconfiguration.

Many approaches to reconfiguration usually rely on a coordinator to manage the reconfiguration logics, and on a distributed framework for the reconfiguration actuation. Referring to Figure 1, the dependability aspects addressed in this paper are the selection (after a decision process) and actuation of a reconfiguration, while the detection of failures and diagnosis is left to specialized tools integrated in the reconfiguration infrastructure.

Our approach is divided in two phases: the *instrumentation phase* and the *execution phase*. The first one is performed before the deployment of the managed application and is in charge of instrumenting the framework with respect to the specific application. During this phase, the developer, looking at the managed application, identifies the critical events (e.g., the components failures) for which system reconfiguration is required, together with the related information that need to be monitored and collected. For each identified critical event, the

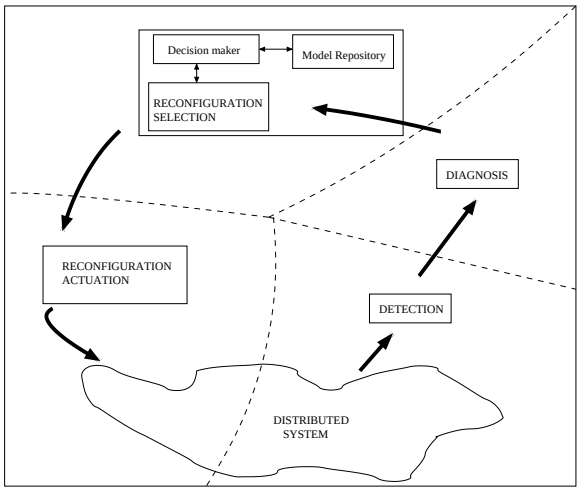

Fig. 1. Dependability Aspects

developer specifies a set of alternative reconfiguration processes, to overcome the event's effects. As an example of reconfiguration process, the developer may define the activation of a component on a different host, the re–connection of a client from its current server to another one, etc. It is important to notice that the definition of the reconfiguration processes depends on the reconfiguration capabilities of the involved components; the components are usually black boxes, thus allowing for reconfiguration only through their public APIs.

The second phase, instead, is performed after the deployment of the managed application, and represents the strategy used by the framework to provide fault tolerance on the managed application. During the execution phase the following operations are performed:

- **Monitoring of the managed system:** monitoring is necessary to collect information about the state of both components and deployment environment, and to detect the critical events within the system. Depending on the components, the critical events can be also notified by a tool which performs error detection and diagnosis.

- **Decision process:** as a reaction to one or more critical events, the system is reconfigured to restore its correct behavior. The decision process, performed by the *Decision Maker*, faces the problem of choosing the new configuration for the managed system, trying to take the most rewarding decision with respect to the detected failure and the current application state. In the presented approach the new configuration is chosen taking into account the feedbacks provided by the online evaluation of stochastic dependability models: in particular, the best one is the configuration which maximizes the health state of the system, as represented by a health function defined fol-

lowing the characteristics of the managed application.

– **Reconfiguration actuation:** once the new configuration is selected, a reconfiguration process places the managed system in that configuration. In this phase, the system which performs the reconfiguration must address several issues, such as the consistency of application data during the reconfiguration process, or the restoration of a correct state. Also in this case, the capability of ensuring consistency during the reconfiguration depends on the ability of the single application components to manage their internal state and, overall, on the degree of control provided to the developer for such management.

3 A framework for fault tolerance provision

This section describes the framework which implements the approach previously described. The framework is based on Lira [9–11], an infrastructure for component based application reconfiguration, enriched with decision making capabilities that allow the choice of the reconfiguration which better fits the actual state of the managed application. In the following, the different parts of the framework, and how they implement the proposed approach are described.

3.1 The Lira infrastructure

Lira (Lightweight Infrastructure for Reconfiguring Applications) is an infrastructure to perform remote monitoring and reconfiguration at component and application level. Lira is inspired to the Network Management [12] in terms of reconfiguration model and basic architecture. The reconfiguration model of the Network Management is quite simple: a network device, such as a router or a switch, exports some reconfiguration variables through an Agent, which is implemented by the device's producer. These variables exported by the agent are defined in the MIB (Management Information Base) and can be modified using the *set* and *get* messages of SNMP (Simple Network Management Protocol) [12].

Using the same idea, in order to manage reconfiguration of software applications, software components play the role of the network devices: a component parameter that may be reconfigured is exported by an agent as a variable, or as a function. Following the architecture of SNMP, Lira specifies three architectural elements: (i) the **Agent**, that represents the reconfiguration capabilities of the managed software components, exporting the reconfigurable variables and functions; (ii) the **MIB** that contains the list of those variables and functions; and (iii) the **Management Protocol**, that allows the communication among the agents.

The Lira agents may be implemented either by the component developer, or by the application assembler: in the last case, the agents can export also functions that implement more complex reconfiguration activities, or variables

that export the result of a monitoring activity performed by the agent on the managed component.

The agents can be hierarchically composed: so, it is possible to define agents managing a sub–system exactly like single components, thus allowing an easier management of the reconfiguration at architectural level. The main advantage of having a hierarchy of agents is the possibility of hiding the reconfiguration complexity behind the definition of a specialized higher level agent. Moreover, this definition favours the scalability of the reconfiguration service, because a reconfiguration at application level is implemented like a reconfiguration at component level. According to this hierarchical structure, in fact, an agent has manager capabilities on the system portion under its own control, while it is a simple actuator with respect to the higher level agents.

A Lira agent is a program that runs on its own thread of control, therefore it is indipendent from the managed components and hosts.

There are four kinds of agents specialized in different tasks. The **Component Agents** (CompAgents) are associated to the software components, they monitor the state of the component and implement the reconfiguration logics. The agent–component communication is component dependent: if a component is implemented in Java and distributed as a *.jar* file, the communication is implemented through *shared memory*. To avoid synchronization problems, the component must provide atomic access to the shared state. If the component provides reconfiguration capabilities through *ad hoc* protocols, the agent is tailored to perform this kind of communication.

The CompAgent manages the life–cycle of the component by exporting the functions *start, stop, suspend, resume* and *shutdown*. The function *shutdown* stops the component and kills the agent. For monitoring purpose, the CompAgent exports the predefined read–only variable *STATUS*, which maintains the current state of the component (started, stopped, suspended). Like every agent, the CompAgent is able to notify the value of a variable to its manager, addressed by the variable *NOTIFYTO*: these variables are defined in every CompAgent MIB.

Host Agents run on the hosts where components and agents are deployed. They represent not only the run time support necessary for the installation and activation of both agents and components, by exporting in their MIB the functions *install, uninstall, activate, deactivate*, but they also allow the monitoring and reconfiguration of the software resources available on the host at operating system level. In fact, by exporting variables and functions associated to the managed resources, the Host Agents allow the reconfiguration of the environment where the application is deployed. As well as the managed host resources, the Host Agent maintains the lists of both installed and activated components, making them available to the manager by exporting in the MIB the variables *ACTIVE_AGENTS* and *INSTALLED_AGENTS*.

The **Application Agent** is a higher level agent which controls a set of components and hosts through the associated agents. This agent manages a sub–system as an atomic component, hiding the complexity of reconfiguration and

increasing the infrastructure scalability. The application agent which controls the whole system is called **Manager**. Figure 2 shows the different kinds of agents.

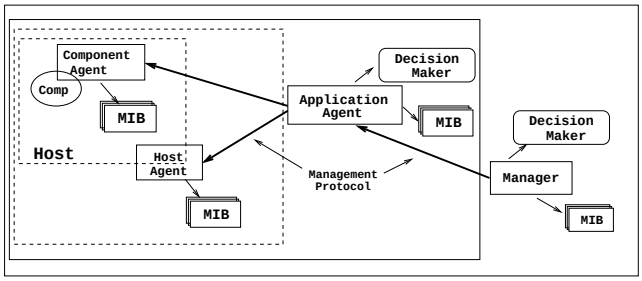

Fig. 2. Lira general architecture

The higher level agents may be programmed by using an interpreted language: the *Lira Reconfiguration Language* [9], which provides a set of commands for the explicit management of reconfiguration on component based applications. The language allows the definition of *proactive* and *reactive actions*: the first ones are continuously performed with a delay expressed in milliseconds, while the latter ones are performed when a specified notification is received by the agent. The reconfigurations performed on the managed components are specified in terms of remote setting of the exported variables or remote call of exported functions. A detailed description of the Lira Reconfiguration Language can be found in [9].

The **Management Protocol** is inspired to SNMP, with some necessary modifications and extensions. Each message is either a *request* or a *response*, as shown in Table 1:

Table 1. Management Protocol messages

request	*response*
SET(*var_name, var_value*)	ACK(*msg_text*)
GET(*var_name*)	REPLY(*var_name, var_val*)
CALL(*func_name, par_list*)	RETURN(*ret_value*)

Requests are sent by higher level agents to lower level ones and responses are sent backwards. There is one additional message, which is sent from agents to communicate an alert at upper level (even in the absence of any request):

NOTIFY(*variable_name, variable_value, agent_name*)

Finally, the **MIB** provides the list of variables and functions exported by the agent which can be remotely managed. It represents the agreement that

allows agents to communicate in a consistent way, exactly like it happens in the network management. Note that also the predefined variables and functions that characterize the different agents (for example, the variable *STATUS* or the function *stop* in the CompAgent) are defined in the MIB, as detailed in [13].

3.2 Lira and the proposed approach

With respect to the previously defined approach, Lira provides the necessary infrastructural support for system monitoring and reconfiguration actuation. In addition, it implements through the agents the interfaces with the external tools used within the framework, such as those for failures detection and for the stochastic models evaluation.

During the instrumentation phase, the developer creates the lower level (component and host) agents by specializing the Java classes of the Lira package. In particular, the host agents must specialize the functions for installation and activation of the components, making them working with the current component based application. Moreover, by using the Lira APIs, the developer defines the variables and functions for remote control and reconfiguration, that will be exported in the agent MIBs. Finally, the agents are programmed to perform local monitoring on the component, and to notify the manager when a critical event (such as a failure) is detected.

The Lira Manager (or Application Agent) implements the monitoring policies, usually specified as a set of *proactive actions*, the reconfiguration policies, usually implemented as a set of *reactive actions* performed when a particular event is received, and the reconfiguration processes, specified as functions implemented using the commands of the Lira Reconfiguration Language. For example:

```
proactive actions
begin
  stateMonitor: every 10000 do
    begin
      string healthState;
      healthState := read(A1, HEALTH_STATE);
      if (healthState = "down")
        then restartNode(N1);
      endif;
    end
end
```

specifies a monitoring activity in which, every 10000 milliseconds, the Manager checks the value of the variable `HEALTH_STATE` exported by the agent A1: if its value is *down*, the node N1 managed by the agent A1 is considered to be not working. In such a case, N1 is restarted, calling the local function `restartNode`. Note that this function can be exported by this agent (actually working as an application agent), making the *restartNode* reconfiguration available to the higher level agents.

In the example just presented, a single reconfiguration process (restarting the node) is specified. In our framework, instead, more sophisticated situations are accounted for, where the occurrence of a single or of a combination of critical events may trigger a number of possible reconfiguration processes, each one implemented by a Lira function. Then, the most rewarding one is selected after a decision process, and put in place by the Manager.

3.3 Decision Maker: general issues

The decision maker (DM) takes decisions about system's reconfiguration, adopting a model-based analysis support to better fulfill this task. In presenting the decision maker subsystem, we outline the main aspects characterizing the overall decision process methodology as well as the involved critical issues.

Hierarchical approach. Decisions can be taken at any level of the agents hierarchy as proposed by Lira and, consequently, the power of the reconfiguration is different. Resorting to a hierarchical approach brings benefits under several aspects, among which: i) facilitating the construction of models; ii) speeding up their solution; iii) favoring scalability; iv) mastering complexity (by handling smaller models through hiding, at one hierarchical level, some modeling details of the lower one). At each level, details on the architecture and on the status of components at lower levels are not meaningful, and only aggregated information is used. Therefore, information of the detailed model at one level is aggregated in an abstract model and used at a higher level. Important issues are how to abstract all the relevant information of one level to the upper one and how to compose the derived abstract models. In our framework the first, bottom level is that of a Component Agent. At this level, the Decision Maker can only autonomously decide to perform preventive maintenance, to prevent or at least postpone the occurrence of failures. An example of preventive maintenance is "software rejuvenation" [14]: software is periodically stopped and restarted in order to refresh its internal state. The second level concerns the Application Agent; the DM's reconfiguration capabilities span all software and hardware resources under its responsibility which encompass several hosts, and installation/activation of new components is allowed. At the highest level there is a Manager agent, which has a "global" vision of the whole system (by composing models representing such subsystems); therefore the DM at this level has the ability to perform an overall reconfiguration.

Composability. To be as general as possible, the overall model (at each level of the hierarchy) is achieved as the integration of small pieces of models (building blocks) to favor their composability. We define composability as the capability to select and assemble models of components in various combinations into a model of the whole system to satisfy specific user requirements [15]. Interoperability is defined as the ability of different models, connected in a distributed system, to collaboratively model a common scenario in order to satisfy specific user requirements. For us, the requirement is to make "good" decisions (in the sense explained later on); models of components are combined into a model of the whole system in a modular fashion and the solution is carried out on the overall

model [16]. Interoperability among models in a distributed environment is not considered at this stage of the work.

For the sake of model composability, we are pursuing the following goals:

- To have a different building block model for each different type of components in the system. All these building blocks can be used as a pool of templates,
- For each component, to automatically (by the decision making software) instantiate an appropriate model from these templates, and
- At a given hierarchical level, to automatically link them together (by rules which are application dependent), thus defining the overall model.

Goodness of a decision. The way to make decisions may be different, depending upon the referred hierarchical level. First of all it has to be defined how to judge the goodness of a decision: actually, the meaning of "good" decision strictly depends on the application domain and so cannot be stated in a general way. However, criteria to discriminate among decisions have to be well-stated since they drive the modeling as well as the decision process. The factors which contribute to make an appropriate decision are the rewards and the costs associated to a decision and its temporal scope. Indeed, costs/benefits balance has to be determined along all the interval from the time a reconfiguration is put in place to the next reconfiguration action. The solution of the overall model has to be carried out rather quickly to be usable. This is an open issue and, if not properly solved, will surely be a limiting factor to the practical utility of the on-line method. In general, to be effective, the time to reach the solution ($T_{Decision}$) and to put it in place ($T_{Actuation}$) has to be much shorter than the mean time between the occurrence of successive events requiring a reconfiguration action ($T_{NextRec}$). The parameter $T_{NextRec}$ may depend on the expected mean time to failure of system components, and on how severe the occurred failure is perceived by the application. Indeed, some failures may require system reconfiguration, while others may not. In general, the criterion for decision taking is a reward function which should account for several factors, including the time to take a decision, how critical is the situation to deal with, and the costs in terms of CPU time.

Suitability of the modeling approaches. An important issue in dependability modeling is the dependency existing among the system components, through which the state of one component is influenced or correlated to the state of others [17]. If it is possible to assume stochastic independence among failure and repair processes of components, the new reconfiguration scheme can be simply evaluated by means of combinatorial models, like fault tree. In case the failure of a component may affect other related components, state-space models are necessary. Since independence of failure events may be assumed only in restricted system scenarios, state-space models are more adequate approaches to model-based analysis in the general case. Therefore, in the proposed framework, each component is modeled with a simple Petri net which describes its forecasted behavior given its initial state. These models are put together and solved by the DM on the basis of the information collected from the subordinated agents.

Dynamic solution. Another important issue to be considered is the dimension of the decision state-space problem. Considering systems which are the result of the integration of a large number of components, as we do in this work, it could be not feasible to evaluate and store offline all possible decision solutions for each case it may happen. These cases are determined by the combination of external environmental factors and internal status of the system (which are not predictable in advance or too many to be satisfactorily managed [18]) and by the topology of the system architecture which can vary along time. In this scenario, online decision making solutions have to be pursued. Although appealing, the online solution shows a number of challenging problems which require substantial investigations. Models of components have to be derived online and combined to get the model of the whole system. Thus, compositional rules and the resulting complexity of the combined model solution (both in terms of CPU time and in capability of automatic tools to solve such models) appear to be the most critical problems which need to be properly tackled to promote the applicability of this dynamic approach to reconfiguration.

3.4 Decision Maker: how it works

In our approach a building block model is associated with each component. A simplified Stochastic Petri Net like model representing a generic building block model is shown in Figure 3. Error propagation effects can be taken into account by this model: e.g. a failure of a component may affect the status of another one. The model of Figure 3 represents only the possible states of a component, omitting all the details (which are implementation dependent) about the definition of the transitions from one state to another (which possibly depend on the status of other components). A token in the place Up means that the component is properly working. A token in one of places $D1$... Dn means that the component is working in a degraded manner (e.g., in the case the component is hit by a transient fault which reduces its functionalities). The places F_{Crash}, F_{Value}, and $F_{Byzantine}$[1] represent the possible ways a component may fail [19]. $Start$ represents the initial state of a component involved in a reconfiguration, to indicate, e.g, that the component is performing installation, restarting, or attempting to connect to another one to be fully operative.

At each level of the decision making hierarchy, the DM has knowledge of the behavior of each system unit visible at the one-step lower level, in terms of the state of the corresponding building block model(s). To make an example, at the manager agent level, the DM has knowledge of the behavior of each component in the system, each one seen as single system unit; in turn, at application agent level, the DM has knowledge of the behavior of each host involved in that application, again each one seen as a single system unit, and so on.

According to the depicted hierarchical reconfiguration process, when an event triggering a reconfiguration action at a certain level occurs, the DM at that

[1] It makes sense having a $F_{Byzantine}$ state only if the component is within a distributed environment and interacts with two or more components.

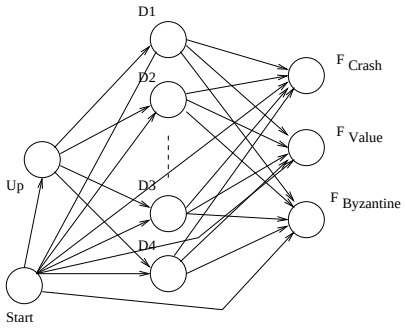

Fig. 3. General building block model

level attempts the reconfiguration, if possible. In case it cannot manage the reconfiguration, it notifies the upper level DM about both the detected problem and its health status. In turn, the upper level DM receiving such request to trigger a reconfiguration, uses such health status information, together with those of the other system units under its control. After taking the decision on reconfiguration at a certain level, the decision is sent to the lower level agents which act as actuators on the controlled portion of the system. Therefore, upon the occurrence of a component failure, the initial states of each host and/or component is retrieved by the application agent by means of its subordinate host and/or component agents. Therefore, the states of any controlled component (provided by Lira) is used as *input* for the appropriate decision maker in the hierarchy. The building block models are then linked together through predefined compositional rules to originate the overall model capturing the system view at such hierarchical level. These compositional rules are applied online to each model component and possibly depend upon the marking of other components, the current topology of the system, and the topology of the system after a reconfiguration.

The information on the topology of the controlled network is stored in a data structure shared between the (application or manager) agent and the decision maker attached to it. The agent uses this information to put in place reconfigurations, while the decision maker uses it to build the online models for forecasting the health status of hosts and/or components participating to a given reconfiguration, and to compare different reconfiguration options. Since the topology of the network can change dynamically as consequence of faults or of a reconfiguration action, new pieces of Petri Nets, representing an host or component, can be changed/removed/added and linked to the overall model. Moreover, statistical dependencies among the different pieces of the overall model and their dynamic changes are captured.

As automatic tool for models resolution, we are using DEEM (DEpendability Evaluation of Multiple-phased systems) [20] which is a dependability modeling and evaluation tool specifically tailored for Multiple-phased systems (MPS). MPS are systems whose operational life can be partitioned in a set of disjoint

periods, called "phases". DEEM relies upon Deterministic and Stochastic Petri Nets (DSPN) [21] as a modeling tool and on Markov Regenerative Processes (MRGP) for the analytical model solution. The analytical solution tool is suitable for decision-making purposes where predictability of the time to decision and of the accuracy of the solution are needed. Actually, the problem of reconfiguring a system can be seen as partitioned in multiple phases, as many as the steps needed for a given reconfiguration policy (e.g. reinstall and restart of a component, opening a new connection between two components).

In accordance with the models structure in DEEM, the model of the whole system is associated with a *phase net* model (PhN). The PhN synchronizes all the steps involved in the reconfiguration: it is composed by deterministic activities which fire sequentially whenever an event associated to a reconfiguration policy happens (see Figure 4). In the PhN only one token circulates and, in general, there exist as many different PhNs as the number of reconfiguration policies.

Fig. 4. General phase net model

Suppose for example that a unit has to be restarted. Initially the component is in state *Start* (see Figure 3), and, upon the firing of the transition of the phase net indicating the expiration of the restart time, the state of the component moves from *Start* to *Up*, or to one of the degraded states, or even to one of the failure states. Thus, decisions are affected by the following factors:

- The current topology of the system;
- The topology of the system after a reconfiguration has been undertaken;
- The steps provided for a reconfiguration policy;
- The current state of each host and/or component;
- The forecasted state of each host and component.

The DM decides the new reconfiguration by solving the overall model and gives back as *output* (to Lira) the *best reconfiguration action*. It is in charge of the Decision Maker to solve such overall composed model as quickly as possible to take appropriate decisions online identifying the most rewarding reconfiguration action among a pool of pre–defined options.

After taking the decision on reconfiguration at a certain level, the decision is sent to the lower level agents which act as actuators on the controlled portion of the system. Obviously, the correctness of the decisions depends on both the accuracy of the models and on its input parameters.

Since both the model-based evaluation and the reconfiguration processes depends on the specific application, the description on how the Decision Maker works cannot go in more details. In order to demonstrate how the proposed

framework actually works, we introduce in the next Section a simple, but effective example, showing the different steps of the intended methodology.

4 A Simple Example: Path Availability in a Communication Network

A simple, but meaningful, scenario is the case of distributed computing where two peer–to–peer clients on the network are communicating. To prevent service interruption, it is necessary to provide an adequate level of paths redundancy among the clients involved in the communication. The network topology we assume consists of six hosts physically (wired) connected as shown in Figure 5a. For management purpose, we consider the network divided in two subnetworks Net_1 and Net_2, which contain the hosts $\{H_1, H_2\}$ and $\{H_3, H_4\}$ respectively. The hosts H_5 and H_6, where the clients are deployed, are not included in the managed network.

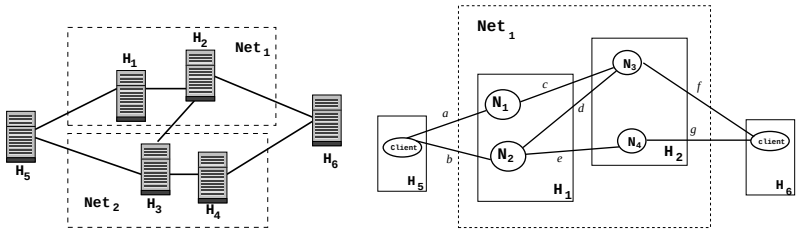

Fig. 5. Hosts physical connection (a) and the logical net (b)

Table 2. Available paths

Path	Route
1	a–N_1–c–N_3–f
2	a–N_1–c–N_3–d–N_2–e–N_4–g
3	b–N_2–e–N_4–g
4	b–N_2–d–N_3–f

A logical communication network composed by logical nodes connected through logical channels is installed on the managed hosts. The nodes $N_1, N_2,\ N_3, N_4$

connected through the channels *a, b, c, d, e, f, g* are deployed on the subnetwork Net_1, as shown in Figure 5b. These channels provide different choices for establishing the communication among the clients, as listed in Table 2.

As said before, the application is managed by the Lira infrastructure: each host H_i is controlled by a Host Agent HA_i, each subnetwork Net_i is controlled by an Application Agent AA_i, while the whole network is controlled by the Manager. The hosts H_5 and H_6 are considered outside the network, so they are not controlled by host agents.

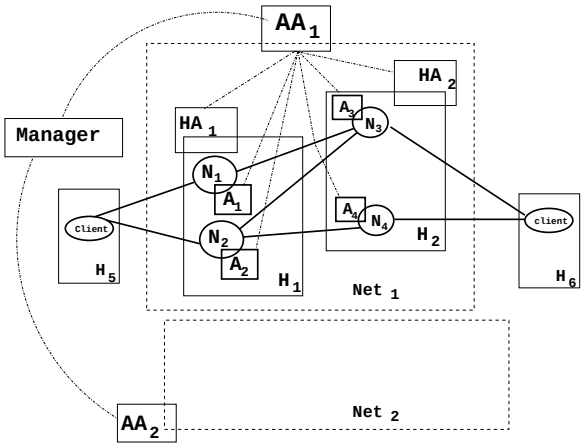

Fig. 6. Lira infrastructure for the controlled network

The logical network is also controlled by the Lira agents. The Component Agents A_i control the logical nodes N_i, and they are managed by AA_1. AA_1 may decide to perform a reconfiguration if it has the necessary information, while it has to ask the general Manager when a global reconfiguration is needed and the local information is not enough. Figure 6 details the Lira management infrastructure.

During the instrumentation phase, the agents A_i and HA_j are programmed to export the enumerated variable *HEALTH_STATE*, which can assume the values *Up, Degraded,* and *Down*, corresponding to the health state of the managed component or host. When this variable is *Down*, the agent is programmed to notify the Application Agent, which will trigger the decision process. The decision process consists in building the models representing the possible new configurations, in evaluating them using DEEM, and in deciding which reconfiguration must be performed to repair the system. During the decision process, only the reconfigurations that place the system in a consistent state are considered and evaluated: the Lira agents are in charge of controlling the application consistency during and after the reconfiguration. It is important to notice that also the costs

of each reconfiguration process are considered during the decision process, to increase the accuracy of the taken decision.

For the proposed example, the paths redundancy can be used to improve the overall availability of the logical network. The goal of the framework is to keep *at least* two paths available between the clients involved in the communication. For the sake of simplicity, we consider that the manifestation of both a hardware fault (such as a wired connection's interruption or a damage in the physical machine) and a software fault (at operating system, application and logical communication level) has a fail-stop semantics, that is the component stops working.

4.1 Measure of Interest and Assumptions

We are interested in monitoring paths availability, so for a path to be available, all the nodes and links in the corresponding route must be available. Note that failures of a particular link or node may result in unavailability of more than one path. For example, if node N_3 fails, paths 1, 2, and 4 become unavailable. To evaluate which of these strategies is the most rewarding we define the "Probability of Failure" of the system as the probability that not even one path exists between the two clients involved in the communication. We analyze this measure both as instant-of-time and as interval-of-time, as function of the time to evaluate which reconfiguration has the lower probability of failure and the lower mean time to failure, respectively. Actually, the instant-of-time measure gives only a point-wise perception of the failure probability representing the distribution function of the failure probability. The interval-of-time measure weights point-wise values over a time interval giving an indication of the mean time to failure. Usually, neither one alone shows a satisfactory indicator and it is interesting to evaluate both of them.

We pursue an analytical transient analysis. The effectiveness of a reconfiguration strategy is studied during an interval of time starting from time T_0 to $T_{NextRec}$. T_0 represents the time at which a failure occurs and the system starts a reaction. $T_{NextRec}$ is the "temporal scope" of the reconfiguration and is an application dependent parameter. Actually, it is useless to test a reconfiguration option after time $T_{NextRec}$.

The framework is instantiated for this case study under the following assumptions:

- Failures occurrence follows an exponential distribution while the time to reinstall, restart or connecting to components follows a deterministic one;
- $T_{NextRec}$ is at least approximatively known;
- T_{Reconf} (that is, $T_{Decision}+T_{Actuation}$) is much less than $T_{NextRec}$ ($T_{Reconf} << T_{NextRec}$).
- No additional failures occur during the interval of time from T_0 to T_{Reconf}. Actually, this assumption is realistic when $T_{Reconf} << T_{NextRec}$.

4.2 Performing reconfigurations

Reconfigurations can be triggered both at AppAgent and Manager Agent levels by their associated Decision Makers. Decisions are taken when a lower level agent notifies that its controlled component is failed. Moreover, to prevent failures of the agent itself, higher level agents proactively ask to the controlled agents for the value *HEALTH_STATE* with a frequency $T << T_{NextRec}$.

As an example of reconfiguration at the AppAgent level, let's suppose that the node N_3 is starting to work in a degraded manner: the associated agent A_3 notifies the variable *HEALTH_STATE* with the value *Degraded*. AA_1 receives the NOTIFY message, and it checks the path availability on the controlled network. Three reconfiguration options are here considered. First, accepting to continue working with node N_3 in a degraded state; this will be referred as "Reduced" later on. A second one is to restart N_3; in this case, the redundancy in terms of paths is temporarily reduced (see Figure 5). This will be referred as "Restart". The third can be to activate a new node N_5 on the host H_2, and to connect it to the client and to the nodes N_1 and N_2, creating new paths. It will be referred as "SetUp".

Obviously the different solutions have different costs in terms of probability of failure, which is the only metric considered in this example to take decisions. It is responsibility of the DM to select the best one.

4.3 Component Building Blocks and Phase Net

The components of the case study can be grouped in two categories: links and nodes. Thus, the corresponding building block models are defined (shown in Figure 7.a and Figure 7.b, respectively). They are quite similar. Each link and node have associated a corresponding Petri net model where only one token circulates. Let's start describing the model of a link, that can be easily generalized to explain that of a component. After an initial possible recovery or set up phase, a generic link X is supposed to be only in state *Up* or *Down* (i.e. a token can be in place $ObjX\_Up$ or $ObjX\_Do$, respectively). When the exponential transition $ObjX\_UpToDo$ fires, the link X fails.

Place $ObjX\_St$ may hold a token only for a while at initial time T_0. A token is in this place if link X, as consequence of system reconfiguration, has to be newly setup or restarted because of a previous failure. From place $ObjX\_St$, a token may move in places $ObjX\_Up$ or $ObjX\_Do$ through the instantaneous transitions $ObjX\_StToUp$ or $ObjX\_StToDo$, to represent the transition of the link X in state *Up* or *Down*, respectively. This allows accounting for a probability of success/failure of the set-up of a new link or of the recovery operation of an old link. Moreover, the time to complete set-up or recovery, through instantaneous transitions $ObjX\_StToUp$ or $ObjX\_StToDo$, depends on the kind of operation. Actually, such transitions are enabled by the Phase Net which emphasizes the length of possible phases following a reconfiguration.

Components are modelled similarly to links. A generic component is shown in Figure 7.b. The only difference is that the possible states of a component

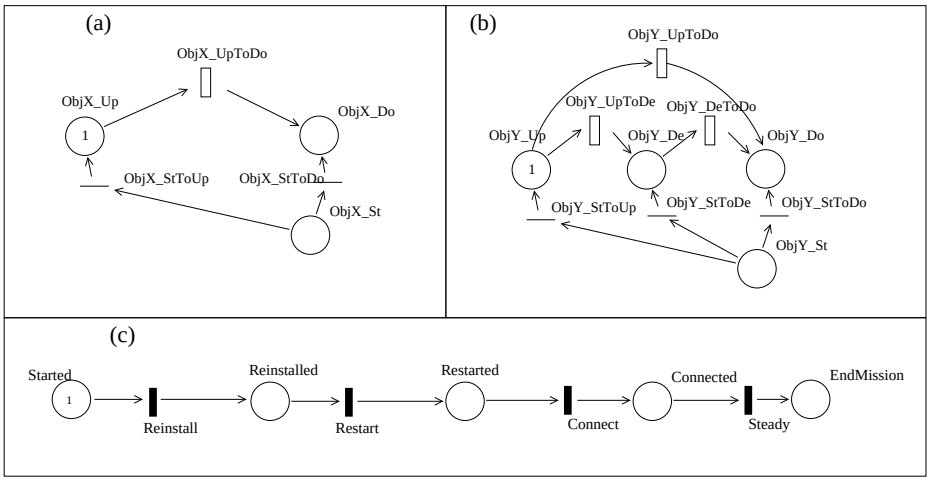

Fig. 7. Building blocks for links (a) and nodes (b), and general phase net model (c)

include also "Degraded". Thus, place $ObjY\_De$ has been added, together with the exponential transitions $ObjY\_DeToDo$ and $ObjY\_UpToDe$ and the instantaneous transition $ObjY\_StToDe$ to connect such place to the rest of the net. This model works analogously to the model of a link, with the difference that a node in the "Up" state goes through "Degraded" before becoming "Down".

A possible Phase Net is shown in Figure 7.c, where a token in place *Started* sequentially moves to place *Reinstalled*, *Restarted*, *Connected*, and then in place $EndMission$.

These DSPN models are specified through the Stochastic Reward Net (SRN) formalism [22]. Beside the general characteristics of SRN, this formalism provides *enabling functions* and *rate/probability functions*[2] which allow to decide, in a marking-dependent fashion, both when a transition is enabled and the rate/probability associated with the firing of such transition. This way, the evolution of a submodel can be dependent on the marking of another one, without explicit and visible links among them. Moreover, the SRN formalism considers the specification and evaluation of the measure of interest as an integral part of the model: reward rates are associated to the states of the models. From this point of view, the tool DEEM we are using allows, also, to i) embed variable parameters in the rate/probability functions, ii) specify the measure of interest in terms of complex reward functions, and iii) to solve the model at varying value of some values of the user-defined variable parameters and/or the time at which the solution is carried out.

In order to clarify how the proposed modeling approach works, Figure 8 shows the overall composed model instantiated for the "Restart" strategy. Simi-

[2] Rate functions are associated to timed transitions, while probability functions to instantaneous transitions

lar considerations hold for the other strategies. The model of Figure 8 is achieved by combining together the models of links and components (on the basis of their respective building block models shown in Figure 7), according to the network topology of Figure 5b and the "Restart" strategy.

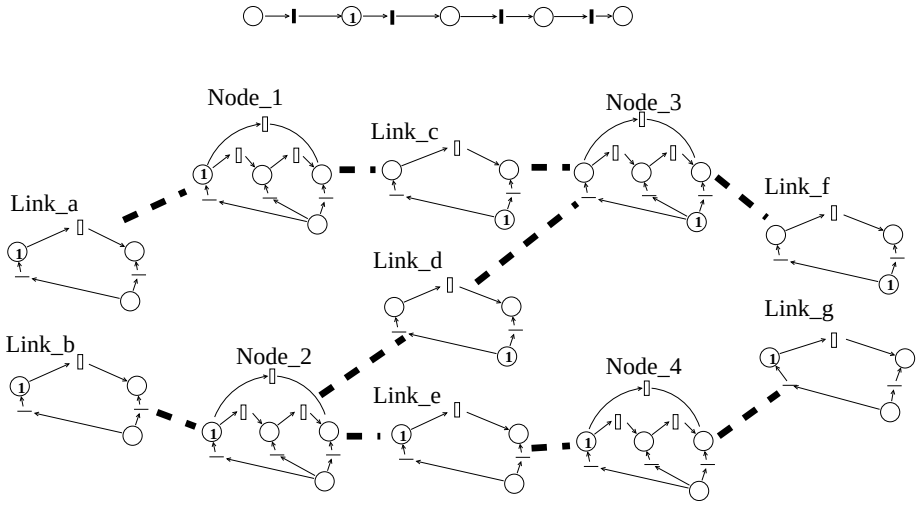

Fig. 8. Overall model for the "Restart" reconfiguration

The "Restart" strategy prescribes that the faulty node N_3 be restarted. In Figure 8 the initial marking of each submodel is indicated by a "1" in the corresponding place. Initially a token is in place *Reinstalled* of the Phase Net (actually the component to be restated is already installed). The initial state of all components (apart the faulty node N_3 and the links c, d, and f connected to it) are initially set according to their $HEALTH\_STATE$ variables (monitored by Lira). The initial marking of node N_3 and its connected links are $ObjN_3\_St$ and $ObjX\_St$ (for X = c, d, and f), respectively. The instantaneous transitions $ObjN_3\_StToUp$, $ObjN_3\_StToDe$, and $ObjN_3\_StToDo$ for node N_3 are enabled only after the firing of the transition *Restart* of the Phase net Model. This synchronization among the involved models is achieved by associating to these transitions the enabling functions "*MARK(Reistalled)==0*" (the function *MARK(Place_Name)* returns the marking of place $Place\_Name$). Similar considerations hold for the instantaneous transitions $ObjX\_StToUp$ and $ObjX\_StToDe$ (for X = c, d, and f). The probability functions of these transitions have to be assigned taking into account that the sum of the probabilities associated to all instantaneous activities connected to the same place have to be one. In the proposed simple example, numerical values have been associated to these probabilities (see Section on setting for numerical evaluation later on). However more complex marking-dependent expression can be assigned, if appropriate.

Notice that the whole model is composed by a set of submodels whose structure is known in advance. In this way, given the building block templates, a set of interdependence rules (which can be derived from the topology of the system or explicitly defined) and the relevant dependability parameters (such as the failure rates and the probabilities involved in the building blocks), the overall model is built automatically by the decision maker and solved by the tool DEEM.

The measure of interest "failure probability" previously introduced is also simply defined and evaluated upon the Petri net model. For example, the failure probability of path 1 of Table 2 is specified as in the following expression:

*(1-MARK(Obja_StToDo)) * (1-MARK(ObjN$_1$_StToDo)) * (1-MARK(Objc_StToDo)) * (1-MARK(ObjN$_3$_StToDo)) * (1-MARK(Objf_StToDo)).*

where each factor represents the probability that the corresponding link or node is operative. In this example, a relatively simple figure of merit has been identified for the analysis. However, the SRN formalism is powerful enough to allow defining and evaluating also performability measures, where dependability factors can be weighted with performance factors and costs.

4.4 Numerical Evaluation

In this Section, the settings for the numerical evaluation and the obtained results are shown. The failure rates of the system components (assuming they are in state "Up") are listed in Table 3, in terms of models transition rates.

Table 3. Default parameters

Parameter	Value	Comments
Observation time interval $T_{NextRec}$	(1200 min.) 20 hours	-
Rate of transition *ObjY_UpToDe*	1e-05 1/min.	for Y = N_1, N_2, N_3, N_4
Rate of transition *ObjY_DeToDo*	1e-04 1/min.	for Y = N_1, N_2, N_3, N_4
Rate of transition *ObjY_UpToDo*	1e-06 1/min.	for Y = N_1, N_2, N_3, N_4
Rate of transition *ObjX_UpToDo*	1e-05 1/min.	for X = a, b, c, d, e, f, g

As consequence of a failure affecting node N_3 the three possible reconfigurations "Reduced", "Restart" and "Set-up" have been evaluated considering the values listed in Table 3, Table 4 and Table 5, respectively. The overall model is almost the same for all the three cases: they change only for their initial marking, enabling conditions and the values of some parameters, as shown in the Tables.

Figure 9 shows the comparison among the different strategies in terms of the failure probability. The instant-of-time measure of the failure probability is the unreliability of the system. The case when all components are "Up" is referred as "Perfect" in the Figure and it represents a lower bound for the failure probability.

Table 4. Strategy "Restart"

Parameter	Value
Component restart time $T_{Restart}$	0.1 min.
Probability that $N3$ restart is successful p_RestUp_N3	0.75
Probability that $N3$ restart is partially successful p_RestDe_N3	0.15
Probability that $N3$ restart fails p_RestDo_N3	0.10

Table 5. Strategy "Set-up"

Parameter	Value	Comments
Component reinstall time $T_{Reinstall}$	0.1 min.	-
Component connection time $T_{Connect}$	0.05 min.	-
Prob. $N3$ reinstall is successful p_ReinUp_N3	0.90	-
Prob. $N3$ reinstall is partially successful p_ReinDe_N3	0.08	-
Prob. $N3$ reinstall fails p_ReinDo_N3	0.02	-
Prob. the new link connection is successful p_ConnUp_X	0.95	for X = c, d, f
Prob. the new link connection fails p_ConnDo_X	0.05	for X = c, d, f

It is interesting to observe that at varying of time the best strategy changes: the strategy that minimizes the failure probability is initially the "Reduced" strategy (up to 4 hours), and, then, the best one becomes "SetUp". This suggests that the choice of the "best" reconfiguration depends also on the mission time for the system, as dictated by the application.

5 Related Work

Recently, several approaches and frameworks to manage non functional properties in component based applications have been proposed in the literature.

In [2], Knight et al. present *Willow*, an architecture for enhancing survivability of critical information systems. Willow is designed to detect *local* and *complex faults* by means of monitoring features, and to react to the failures by dynamically reconfiguring the managed system. The broad and complete management of reconfiguration in terms of component dependencies, conflicting policies, global data and state consistency, makes Willow a suitable framework for survivability provision of large scale distributed applications. However, the Willow framework results quite heavy and appropriate for applications composed by hundreds of thousands components. For usual distributed systems, where components are in the order of tens, this framework can be too expensive to instrument and deploy, and many of its functionalities may remain unused.

In [1, 23] Garlan et al. present an approach to dependability and performance management based on *monitoring, interpretation* and *reconfiguration*, where the

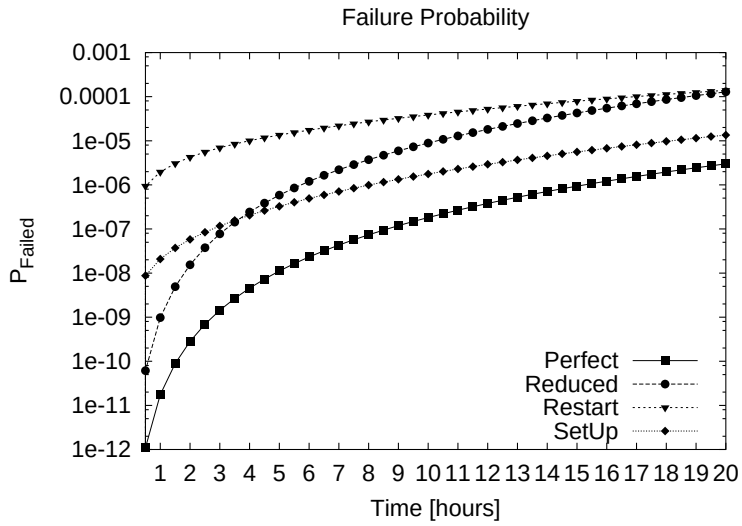

Fig. 9. Comparison among different choices

Software Architecture plays a central role. The Software Architecture of the managed system is specified in terms of components and connectors using Acme [24]. Then, the architectural model is enriched by using the runtime information provided by the *gauges* and finally it is evaluated by using AcmeStudio. If the current architecture violates the specified constraints, it is dynamically modified by the framework. In order to react to the detected failures, a set of *repair strategies* are associated to a constraint violation, while a set of *tactics* is carried out to actually repair the managed system. The tactics rely on some *adaptation operators*, defined by the developer within the framework taking into account the actual reconfiguration capabilities of the managed components. This last approach is very similar to our reconfiguration based approach: in fact, the abstraction of data performed by the gauges is provided in Lira by a hierarchy of agents, while the repair strategies and tactics are defined using the reconfiguration operators provided by the Lira Reconfiguration Language. In Lira, the higher level agents must be programmed to explicitely manage the architectural constraints, a task performed in [1] by the AcmeStudio tool by simply defining the architectural family in Acme ADL. In general, the instrumentation phases of both approaches depend on the complexity of the managed applications: for systems where the types of components are limited in number and the architecture is not extremely complex, the framework proposed in this paper should be instrumented quicker than the other.

With respect to the decision making process, both the approaches proposed in [1] and in [2] do not provide a way to choose the new system configuration specifically created for the optimization of dependability properties, namely fault tolerance. The choice of the tactics in [1] and the *postures* in [2] which place the

system in a new configuration are not driven by dependability concerns: in fact, they are just chosen among a set of the possible ones which repair the system. In our approach, instead, among the possible reconfigurations it is chosen the one that places the system in a more healthy state, taking into account the feedbacks provided by the evaluation of Stochastic Timed Petri Nets built at run time to represent the current topology and state of the managed systems.

6 Conclusions

This work has proposed a framework for reconfiguration-based fault tolerance in distributed systems. It integrates Lira, a light-weight infrastructure for reconfiguring applications, with a model–based Decision Maker. On the basis of the monitoring activity carried on by the Lira agents, the Decision Maker performs system reconfiguration in presence of components malfunctions. It resorts to an on-line evaluation of the different available reconfiguration strategies in order to select the most profitable one, in accordance with some predefined optimization criteria. Building block models for the different typology of system components (namely, node and link), capturing the correct/faulty behavior of the components they are associated to, are instantiated on line and connected to reflect the current system configuration.

Since the evaluation process and the system reconfiguration are application dependent, the methodology is firstly described in a very general way, then it is detailed using a simple case study. Managing the communication between two clients in a peer–to–peer network, we have shown how the system is monitored and reconfigured to maintain the communication capabilities, and overall how the framework is able to take online decisions by dynamically setting up and solving stochastic models from available general building blocks.

Applying reconfigurations which adapt to evolving systems structures is highly attractive nowadays, and methodologies pursuing such objective are therefore very appealing. The work we have presented constitutes a relevant initial step in this direction. However, at the current stage of definition, a number of open issue and possibly limiting factors are left unresolved, and further investigations are necessary. The identification of appropriate compositional rules to dynamically build up the overall system model, of the criteria to base the reconfiguration selection process on and, primarily, of the necessary features to manage the complexity of the model solution in order to make the on-line evaluation a feasible approach, are among the most challenging problems still under investigations.

In addition, to provide an effective means to fault tolerance in critical systems, the Lira infrastructure itself has to be made fault–tolerant. Another possible research direction is to improve error diagnosis capabilities of each agent, to better calibrate system reconfiguration.

Acknowledgments. This work has been partially supported by the Italian Ministry for University, Science and Technology Research (MIUR), project “Stru-

menti, Ambienti e Applicazioni Innovative per la Societa' dell'Informazione, SOTTOPROGETTO 4".

References

1. Garlan, D., Cheng, S.W., Schmerl, B.: Increasing System Dependability through Architecture-based Self-repair. In: Architecting Dependable Systems. Springer-Verlang (2003)
2. Knight, J.C., Heimbigner, D., Wolf, A.L., Carzaniga, A., Hill, J., Devanbu, P., Gertz, M.: The Willow Architecture: Comprehensive Survivability for Large-Scale Distributed Applications. In: International Conference of Dependable Computer and Systems (DSN02), Washington DC (2002)
3. Kramer, J., Magee, J.: Dynamic Configuration of Distributed System. IEEE Transaction of Software Engineering **SE** (1985) 424–436
4. Kramer, J., Magee, J.: The Evolving Philosophers Problem: Dynamic Change Management. IEEE Transactions on Software Engineering **16** (1990) 1293–1306
5. Young, A.J., Magee, J.N.: A Flexible Approach to Evolution of Reconfigurable Systems. Proc. of IEE/IFIP Int. Workshop on Configurable Distributed Systems (1992)
6. Magee, J.: Configuration of Distributed Systems. In Sloman, M., ed.: Network and Distributed Systems Management. Addison-Wesley (1994)
7. Kramer, J., Magee, J.: Analysing Dynamic Change in Software Architectures: A Case Study. Proc. 4th Int. Conf. on Configurable Distributed Architecture (1998) 91–100
8. Wermelinger, M.: Towards a Chemical Model for Software Architecture Reconfiguration. Proceedings of the 4th International Conference on Configurable Distributed Systems (1998)
9. Castaldi, M., De Angelis, G., Inverardi, P.: A Reconfiguration Language for Remote Analysis and Application Adaptation. In Orso, A., Porter, A., eds.: Proceedings of Remote Analysis and Measurement of Software Systems. (2003) 35–38
10. Castaldi, M., Carzaniga, A., Inverardi, P., Wolf, A.: A Light-weight Infrastructure for Reconfiguring Applications. In Westfechtel, B., van der Hoek, A., eds.: Software Configuration Management, ICSE Workshops SCM 2001 and SCM 2003 Toronto, Canada, and Portland, OR, USA. Volume 2649 of Lecture Notes in Computer Science., Springer (2003)
11. Castaldi, M., Costantini, S., Gentile, S., Tocchio, A.: A Logic-based Infrastructure for Reconfiguring Applications. Technical report, University of L'Aquila, Department of Computer Science (2003) To appear in LNAI, Springer.
12. Rose, M.T.: The Simple Book: An Introduction to Networking Management. Prentice Hall (1996)
13. Castaldi, M., Ryan, N.D.: Supporting Component-based Development by Enriching the Traditional API. In: Proceedings of Net.Object Days 2002 - Workshop on Generative and Component-based Software Engineering, Erfurt, Germany (2002) 44–48
14. Huang, Y., Kintala, C., Kollettis, N.: Software rejuvenation: Analysis, module and applications. In: Proc. of 25th Int. Symposium on Fault-Tolerance Computing (FTCS-25), , Pasadena, CA, USA (June 1995)
15. Petty, M.D., Weisel, E.W.: A Composability Lexicon. In: Proceedings of the Spring 2003 Simulation Interoperability Workshop, Orlando FL, USA (2003)

16. Betous-Almeida, C., Kanoun, K.: Stepwise Construction and Refinement of Dependability Models. In: IEEE International Conference on Dependable Systems and Networks, Washington D.C, USA (2002)
17. Siewiorek, D.P., Swarz, R.S.: Reliable Computer System - Design and Evaluation. 3 edn. Digital Press (2001)
18. Chohra, A., Porcarelli, S., Di Giandomenico, F., Bondavalli, A.: Towards Optimal Database Maintenance in Wireless Communication System. In: 5th World Multi-Conference on Systemics, Cybernetics and Informatics, (SCI2001), Orlando, Florida (2001)
19. Powell, D.: Failure Mode Assumptions and Assumption Coverage. In Laprie, J., Randell, B., Kopetz, H., Littlewood, B., eds.: Predictably Dependable Computing Systems. Springer Verlag (1995) 3–24
20. Bondavalli, A., Mura, I., Chiaradonna, S., Filippini, R., Poli, S., Sandrini, F.: DEEM: a Tool for the Dependability Modeling and Evaluation of Multiple Phased Systems. In: Proc. of Dependable Systems and Networks, New York, USA (2000)
21. Marsan, M.A., Chiola, G.: On Petri Nets with Deterministic and Exponentially Distribuited Firing Times. In: LNCS. Volume 266. Springer Verlang (1987) 132–145
22. Muppala, A.K., Ciardo, G., Trivedi, K.S.: Stochastic reward nets for reliability prediction. Communications in Reliability, Maintenability and Serviceability **1** (1994) 9–20
23. Garlan, D., Schmerl, B., Chang, J.: Using Gauges for Architecture-Based Monitoring and Adaptation. In: Proceedings of Working Conference on Complex and Dynamic Systems Architecture, Brisbane, Australia (2001)
24. Garlan, D., Monroe, R., Wile, D.: Acme: Architectural Description of Component-Based Systems. In Leavens, G.T., Sitaraman, M., eds.: Foundations of Component-Based Systems. Cambridge University Press (2000) 47–68