Automatic Configuration with *Conflets*

Justinian Oprescu, Franck Rousseau, and Andrzej Duda

LSR-IMAG Laboratory
Grenoble – France
{Justinian.Oprescu, Franck.Rousseau, Andrzej.Duda}@imag.fr
www-lsr.imag.fr

Abstract. In this paper, we introduce a framework for the automatic configuration of applications running in dynamic environments where changes are frequent. We propose a way to describe, for each application, its configuration policy, and the execution environment's factors that affect its behavior. On this basis, we can generate application-specific configuration tools, called *conflets*. The application's source code is not required. The conflet combines the information drawn from the execution environment with the knowledge of the configuration policy. It is therefore able to detect when and how the execution environment modifies its characteristics, and can automatically react by reconfiguring the application and thus adapting it to the dynamic environment.

1 Introduction

Starting an application can be as simple as clicking an icon, or typing a sequence of characters in a terminal window. However, most of the time, several other operations must be fulfilled. Indeed, the applications require various configuration operations, such as provision of resources, editing of configuration files, or deployment of dependencies.

Under normal conditions, users can deal with configuration. Quite often, they are as well assisted by configuration assistants or wizards included by most of the modern operating systems or large applications. These assistants hide the complexity of the configuration, but are unaware of the dynamic nature of the execution environment. Their weakness lies in the interactive design, which supposes that the user can always initiate the configuration, and provide whatever information is needed. While not an issue in traditional computing environments evolving at human-speed, this missing feature becomes important in dynamic environments where changes are frequent, and potential interaction windows between applications are small. Hermann et al. evoke several mobile scenarios showing that, in dynamic environments, one of the most important resources is time [1]. Tennenhouse shares the same opinion, and predicts that the growing number of intelligent entities operating at faster-than-human speed pushes the human-in-the-loop computing to its limits [2]. More dynamic the environment is, less time remains between two successive modifications. Below a threshold of order of tens of seconds, it is clear that users cannot interfere anymore, as the rate of

modifications is too important. Any configuration activity going on for too long might be obsoleted before ending by the next occurring change. Kebhart and Chess see self-configuration as the first step toward autonomic-computing [3]. They explain that once the systems are able to configure automatically, we may look forward to self-optimization, self-healing, and self-protection as next milestones on the road to building systems that can manage themselves.

In this paper, we deal with automatic configuration for the applications running in dynamic environments. We introduce the concept of sequential configuration pattern that states that, for any application, the configuration can be divided into several sequential operations. We describe them in the Application Configuration Policy (ACP). During these operations, various interactions with the execution environment take place. We describe them in the Environment Awareness Policy (EAP). While the ACP focuses on the application's universe (like the existent configuration assistants), the EAP deals with the dynamism of the environment. We coined the name of *conflet* for configuration tools that implement both policies, being thus able to detect changes occurring in the environment, and to automatically react by adapting the application behavior. In this paper, we present the conflet architecture, discuss several design issues, and analyze the performances of our implementation.

2 Configuration Patterns

With respect to the application starting time, we identify three moments when configuration can be performed.

Before execution Parameters that can be set-up before the application starts are usually persistent, typically stored on disk in configuration files.
At start-up time Command-line parameters are provided by users or scripts.
At runtime Runtime parameters are usually interactively provided to the application via a user interface.

All parameters are handled only during execution, and each one has a predetermined effect on the application, the order in which they are processed being less important. Usually, the persistent and the command-line parameters are processed only once, at start-up. Applications that allow only these two ways of configuration are said to follow a two-step configuration pattern. Applications that additionally allow runtime configuration follow a three-step configuration pattern. Both patterns are sequential, as shown on Figure 1, and we consider that most applications follow such a model.

Although a given parameter might be set-up in different ways, its effect is always the same. For instance, the URL of the document to be displayed in a web browser might be given in a persistent way as the default start-up page, but can be overridden by a URL provided on the command-line, or at runtime through the user interface. Even if the configuration time is different (before execution, at start-up, or at runtime), the effect of setting the parameter is the same: the browser loads and shows the document at the specified URL.

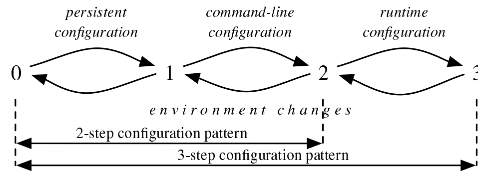

Fig. 1. Sequential configuration pattern

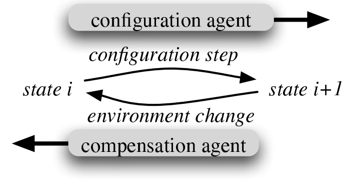

Fig. 2. Compensation agent

Nevertheless, the time at which the configuration of a specific parameter might take place is important, because it will affect the reconfiguration process. An application supporting only persistent and command-line parameters will have to be restarted for reconfiguration, whereas applications supporting runtime modification of their parameters won't need it, hence a shorter (re)configuration time, and fewer resources used. The sequential nature of the configuration patterns guarantees that minimizing configuration time amounts to minimizing the number of configuration steps as these operations are performed sequentially.

Configuration parameters provide information on the execution environment of an application, hence each configuration step leads the application closer to a state where its set-up is coherent with its environment. However, this set-up might get in an incoherent state after a change in the execution environment, thus requiring reconfiguration. The configuration state to which an application should return after an *environment change* is determined by the number of configuration steps needed to set up the modified parameters. For instance, a mailer for which the SMTP server can be given only through it's configuration file (persistent configuration) has to be restarted to cope with any environment change affecting this server, hence returning to state 0 of the process (see Figure 1). If runtime configuration had been an option, going back to state 2 would have been more efficient, avoiding restart.

Computing the shortest configuration sequence requires the knowledge of the parameters that have changed, i.e. needing reconfiguration, and the state in which their configuration is possible. To solve these issues we propose the Application Configuration Policy (ACP) and the Environment Awareness Policy (EAP), presented in the remainder of this section.

Application Configuration Policy

The Application Configuration Policy (ACP) describes the configuration operations. Each one is performed by a configuration agent that executes configuration commands, and uses a set of resources (parameters) provided by the execution environment.

Each configuration agent has a counterpart in the form of a compensation agent that can undo its actions, thus returning in a previous state (see Figure 2). The compensation agents have a roll-back effect. When the application is recon-

figured, they bring it in the state from where the shortest configuration sequence (performed by configuration agents) can start.

The ACP specifies a list of parameters for which configuration agents need values as inputs. The Environment Awareness Policy defines how to get them.

Environment Awareness Policy

The Environment Awareness Policy (EAP) define how to gather up-to-date values for the configuration parameters. These parameters are resources provided by the execution environment and needed by the application, and whose modification should trigger reconfiguration, using the configuration agents specified by the ACP.

For each parameter, the EAP lists the appropriate tools that must be used to detect changes, and to fetch the most recent value. They are called sources, and for a given parameter, they are considered equivalent, as they implement different ways to obtain the same value, but have priorities defining the order in which they are inquired. For example, the parameter for an SMTP server might have three distinct sources, one relying on DHCP, the second using a service discovery protocol, the third asking input to the user.

By combining the ACP and EAP, we can detect when and how the execution environment modifies its characteristics, infer the state in which the application must return in order to be properly reconfigured, bring the application in that state by applying compensations, and finally perform the configuration. We built the configuration framework in order to address these issues.

3 Configuration Framework

The configuration framework is a generic piece of software that takes as entries an Application Configuration Policy (ACP) and the correspondent Environment Awareness Policy (EAP), and generates a conflet specific to the application. It was first developed as part of the OmniSphere project that aims at developing personalized communication spaces offering the user new applications built on-the-fly by data-flow composition from dynamically discovered services [4–6]. The conflet does not depend on the programming language of the application, and uses operating system standard facilities, such as process or file handling, implemented within configuration and compensation agents. We chose to implement the framework in Java, and to code both policies in XML.

Conflet Architecture

A conflet has a tree-like structure. Figure 3 shows the conflet of an application that follows a three-step configuration pattern. The top element is the scheduler. It governs the execution of configuration and compensation agents. Agents are generated on-the-fly by agent factories whenever the value of one of subjacent

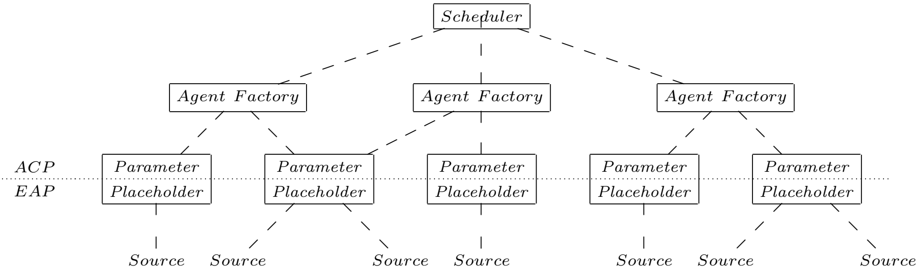

Fig. 3. Conflet architecture for a three-step configuration pattern

parameters changes. Up-to-date parameter values are produced by parameter sources embedded in the environment.

The implementation of each component of the conflet depends on the application to be configured. However the basic behavior is provided through generic classes that can be specialized. The generic architecture follows the pipes-and-filters pattern, receiving event via an entry queue, performing its task, and notifying other registered components about the result of this task.

Parameter Placeholders The sources send modifications events each time they detect a specific modification in the environment. The parameter placeholders listen for such events, analyze the conveyed modification, and decide whether to propagate it or not.

Low-Pass Filters Allowing the components of the conflet to send notifications too often is an open door for denial-of-service attacks. Therefore, the parameter placeholders, and the agent factories implement output low-pass filters that prevent the components downstream from flooding or bursts. A filter behaves like a leaky bucket that can hold at most one item (or event). Although a typical leaky bucket would discard any item arriving when another is already in, the low-pass filter keeps the arriving item and discards the other. Each filter has a sending frequency. At each tick, the event in the bucket (which represents the last modification) is retrieved and transmitted to registered listeners. Ignoring several modifications that occur too quickly has no negative effect, as the agent factories and the scheduler (the components that receive modifications events) do not depend on the modification events history.

Agent Factories The role of the agent factories is to gather up-to-date values from all managed parameters, and to generate agents. The factories receive modification events from parameters and, in response, generate new pairs of agents (configuration and compensation agents). An agent is initialized with the list of current parameter values, and a set, called Δ, of references to the parameters whose values have changed since the last agent passed through the factory low-pass filter.

Scheduler Both new configuration and compensation agents are not executed immediately after creation. The exact moment of execution and their relative order is given by the conflet scheduler that has a global view of the

application state. For example, a configuration agent that starts an application, cannot be executed if the application is already started. Before, the scheduler must roll-back the current configuration by executing necessary compensation agents. In the next section, we discuss the scheduling algorithm that decides the agents to be executed and their order.

Scheduling Algorithm

To manage the application state, the scheduler uses a compensation stack. Every time a configuration agent is executed, the corresponding compensation agent is pushed into the stack. When the execution environment changes and obsoletes the current configuration, one or more compensation agents are retrieved from the stack and executed before the new configuration is applied. The last-in, first-out property of the stack guarantees that the compensations are performed in reverse order. The number of compensations that should be retrieved from the stack and executed depends on the new configuration and is calculated by the scheduling algorithm.

The algorithm uses a list containing the last generated agent for each agent factory. When a new agent is provided to the scheduler, it replaces, in the list, the previous agent generated by the same agent factory. As a new agent means that the application must be reconfigured, the scheduler must then decide what compensation and configuration agents should execute. It calculates therefore the shortest configuration sequence.

First, it computes the global Δ set for all agents, as the union of Δ sets of all agents in the list:

$$\Delta = \Delta_1 \cup \Delta_2 \cup \cdots \cup \Delta_n.$$

The global Δ set contains references to all parameters whose values have changed since the last configuration, and needs to be addressed by the current one. Executing all agents surely configures them all, and hence represents a valid method of reconfiguration. However, it implies as well that the longest possible sequence of configuration agents is executed. And because different agents can configure the same parameter (that is, a parameter can appear in the Δ set of several agents), there may be a shorter sequence with the same Δ set. The scheduling algorithm then calculates m, with the following properties:

$$\Delta = \bigcup_{i=m}^{n} \Delta_i, \ m \in \{1 \ldots n\}, \ m \text{ is maximal}$$

Because it is maximal, m gives the minimal number of agents $(n - m + 1)$ that must be executed in order to reconfigure all parameters in the global Δ set. It also represents the number of compensation agents to be retrieved from the compensation stack and executed before. As configuration agents m to n are executed, their compensation agents are pushed into the compensation stack.

The scheduling algorithm calculates a sequence of agents, which reconfigures all parameters referenced in the global Δ set. The maximum property of m guarantees that the result is the shortest valid sequence. Additionally, because of the sequential nature of the configuration pattern, all the other valid

sequences would be longer and would include the calculated sequence. Therefore, the scheduling algorithm indirectly minimizes the length and the resources used by the configuration.

XML Policies

The configuration framework generates an application conflet on the base of information provided by the ACP and EAP policies. Both are declarative and grouped in a single XML document. We have defined the following DTD.

All the components of a conflet (scheduler, agent factories, parameter placeholders, and parameter sources) are represented by Java classes, hence we need a serialization mechanism to marshal and unmarshal their state to and from XML. Currently we support Jox serialization[1].

```
<!ELEMENT conflet (EAP, ACP, state)>
<!ELEMENT state (serial?, ANY?)>
<!ATTLIST state
    class CDATA #REQUIRED  <!-- URL: Java class to be loaded. -->
>
<!ELEMENT serial (#PCDATA)> <!-- URL of an external file containing the
    object serialization. If empty, the serialization follows inline. --
    >
<!ATTLIST serial
    tool CDATA #REQUIRED  <!-- The serialization mechanism: "jox". -->
>
```

The XML element named state contains the serialization. The Java class is dynamically loaded, as its URL is provided as an attribute of the state element. The object is created using the no-arguments constructor, and can be further initialized using the serialization mechanism.

EAP The Environment Awareness Policy defines the list of parameters and, for each one, a unique identifier and a list of sources. The state element contains the serialization of every Java object.

```
<!ELEMENT EAP (parameter)*>
<!ELEMENT parameter (source+, state)>
<!ATTLIST parameter
    ref ID #REQUIRED <!-- The unique parameter identifier. -->
>
<!ELEMENT source (state)>
```

ACP The Application Configuration Policy contains the list of agent factories, and the agents scheduler. For each agent factory, a unique identifier and the list of parameter references are provided. Each reference uniquely identifies a previously defined parameter.

```
<!ELEMENT ACP (factory+, scheduler)>
<!ELEMENT factory (pref*, state)>
<!ATTLIST factory
    ref ID #REQUIRED <!-- The unique factory identifier. -->
>
<!ELEMENT pref EMPTY>
<!ATTLIST pref
    id IDREF #REQUIRED <!-- A reference to a parameter. -->
>
```

[1] http://www.wutka.com/jox.html

The scheduler manages the order in which the agents are executed. It is therefore given the sorted list of agent factories. The order is important as it is the order in which the generated agents will be executed.

```
<!ELEMENT scheduler (fref+,state)>
<!ELEMENT fref EMPTY>
<!ATTLIST fref
    id IDREF #REQUIRED <!-- A reference to an agent factory. -->
>
```

Performance Issues

We ran several experiments in order to validate our proposal: we measured (i) the time taken by the conflet to produce valid configuration sequences, (ii) the conflet generation time, and (iii) its memory consumption. We carried out all experiments on an Intel Pentium M 735 1.7 GHz laptop with 512 MB of memory running Windows XP and the Java Virtual Machine 1.5 as well as an Intel Pentium III 650 MHz laptop running Linux Fedora Core 3 and the JVM 1.5. All the results represent the average of at least 10 trials.

The measured time between the time a source detects an *environment change* and the time the configuration sequence is ready to be executed is insignificant. For conflets with less than 10 agent factories and with low-pass filters that do not introduce any delays (their frequency is greater than the rate of modifications), observed time was less than 1 ms.

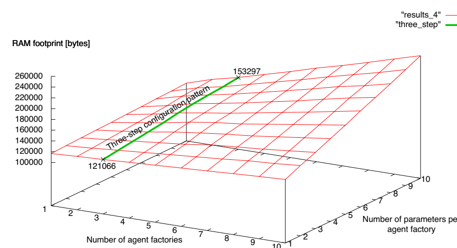

Fig. 4. RAM footprint of conflets

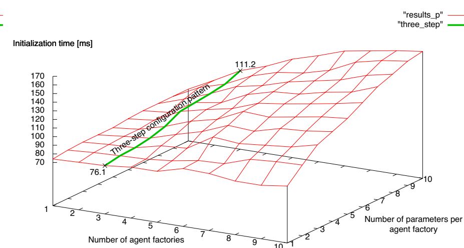

Fig. 5. Conflet generation time

The conflet RAM footprint varies with the number of agent factories, and with the number of parameters per factory, as it is shown on Figure 4. The footprint of a typical three-step conflet is between 121066 and 153297 bytes. Note that only the Java object heap was considered here, thus ignoring the memory occupied by Java code and thread stacks.

Figure 5 shows the conflet generation time (parsing the XML file, and creating all Java objects) according to the number of agent factories and the number of parameters per factory. Generation times are under 200 ms in all cases. However, generating huge conflets (50 agent factories with 50 parameters each) can take almost 3.5 seconds.

4 Related Work

Automatic configuration needs to embrace two important issues: systems need to detect changing environmental conditions or changing system capabilities and to react appropriately. The ability to deal with the dynamism of the execution environment is the principal concern of service discovery protocols [7–10]. Within our configuration framework such tools can be wrapped by parameter sources, and used to detect changes, such as services that are added, or removed from the network.

While service discovery protocols are considered well suited for ad-hoc environments, DHCP [11] is preferred in administered environments. DHCP allows clients to obtain configuration information from a local server manually initialized by an administrator. Akin to service discovery protocols, DHCP clients can be wrapped by parameter sources.

Self-configuration is addressed by several projects that, still, focus mainly on component-based applications. Fabry explained in 1976 how to develop a system in which modules can be changed on the fly [12]. Henceforth, lots of projects dealing with the modification of interconnections between components appeared [13–15]. Such approaches can be employed by configuration agents specific to component-based applications.

The Harmony project, proposed by Keleher et al. [16, 17], allows applications to export tuning alternatives to a higher-level system, by exposing different parameters that can be changed at runtime. Although Harmony and the conflet framework share the same vision, the former focuses on performance tuning and requires that applications are "Harmony-aware". Conversely, the conflet framework can configure applications that were not designed for automatic configuration and whose source code is not available.

5 Conclusion

In this paper, we presented a configuration framework that allows applications to be automatically (re)configured. Automatic configuration is made possible by separating the information about the configuration in two complementary classes. The first describes the application configuration policy, while the second details the external factors that influence the application. The conflet combines both: it monitors the execution environment and reacts to the modifications of external factors by reconfiguring the application accordingly. An external tool, the conflet, that is automatically generated from an XML description, manages the application's execution to adapt it to the dynamism of the execution environment.

A significant advantage of separating the configuration policies from the execution environment awareness, and externalizing them into XML, is the ability to add new detection tools without modifying the application.

References

1. Hermann, R., Husemann, D., Moser, M., Nidd, M., Rohner, C., Schade, A.: DEAPspace – Transient Ad Hoc Networking of Pervasive Devices. Computer Networks **35** (2001) 411–428
2. Tennenhouse, D.: Proactive Computing. Comm. of the ACM **43** (2000) 43–50
3. Kephart, J., Chess, D.: The Vision of Autonomic Computing. IEEE Computer Magazine **36** (2003) 41–50
4. Rousseau, F., Oprescu, J., Paun, L.S., Duda, A.: Omnisphere: a Personal Communication Environment. In: Proceedings of HICSS-36, Big Island, Hawaii (2003)
5. Oprescu, J., Rousseau, F., Paun, L.S., Duda, A.: Push Driven Service Composition in Personal Communication Environments. In: Proceedings of PWC 2003, Venice, Italy (2003)
6. Oprescu, J.: Service Discovery and Composition in Ambient Networks. PhD thesis, Institut National Polytechnique Grenoble (2004) in french.
7. Guttman, E., Perkins, C., Veizades, J., Day, M.: Service Location Protocol, Version 2. IETF RFC 2608, Network Working Group (1999)
8. Jini Community℠: Jini™ Architecture Specification (2005) `http://www.jini.org/standards`.
9. UPnP Forum: UPnP™ Device Architecture 1.0 (2003) Version 1.0.1 `http://www.upnp.org/resources/documents.asp`.
10. Cheshire, S., Krochmal, M.: DNS-Based Service Discovery. IETF draft (2004) Expires August 14, 2004.
11. Droms, R.: Dynamic Host Configuration Protocol. IETF RFC 2131, Network Working Group (1997)
12. Fabry, R.: How to design a system in which modules can be changed on the fly. In: 2nd Intl Conf. on Software Engineering. (1976)
13. Plasil, F., Balek, D., Janecek, R.: SOFA/DCUP: Architecture for Component Trading and Dynamic Updating. In: Proceedings of ICDCS 1998. (1998)
14. De Palma, N., Bellissard, L., Riveill, M.: Dynamic Reconfiguration of Agent-based Applications. In: The European Research Seminar on Advances in Distributed systems (ERSADS). (1999)
15. Batista, T., Rodriguez, N.: Dynamic Reconfiguration of Component-Based Applications. In: Intl Symp. on Software Engineering for Parallel and Distributed Systems. (2000)
16. Keleher, P., Hollingsworth, J.K., Perkovic, D.: Exploiting Application Alternatives. In: Proceedings of ICDCS 1999. (1999)
17. Ţăpuş, C., Chung, I.H., Hollingsworth, J.: Active Harmony: Towards Automated Performance Tuning. In: Proceedings of SuperComputing. (2002)