Pandora: An Efficient Platform for the Construction of Autonomic Applications

Simon Patarin[1] and Mesaac Makpangou[2]

[1] University of Bologna,
Computer Science Department,
Bologna, Italy
patarin@cs.unibo.it
[2] INRIA Rocquencourt,
Regal Group,
Rocquencourt, France
mesaac.makpangou@inria.fr

Abstract. Autonomic computing has been proposed recently as a way to address the difficult management of applications whose complexity is constantly increasing. Autonomic systems will have to diagnose the problems they face themselves, devise solutions and act accordingly. In consequence, they require a very high level of flexibility and the ability to constantly monitor themselves. This work presents a framework, Pandora, which eases the construction of applications that satisfy this double goal. Pandora relies on an original application programming pattern — based on stackable layers and message passing — to obtain a minimalist model and architecture that allows control of the overhead imposed by the full reflexivity of the framework. A prototype of the framework has been implemented in C++, freely available for download on the Internet. A detailed performance study is given, together with examples of use, to assess the usability of the platform in real usage conditions.

1 Introduction

Large-scale distributed applications are being more and more used. Content-delivery networks, computing grids, peer-to-peer file-sharing systems, distributed hash tables, ubiquitous systems: there are many examples and the list keeps growing. The environment in which these applications are deployed, Internet, is characterized by its heterogeneity, the rapid evolution of its various components (hardware, software, but also human) and its lack of reliability. The diversity of the platforms makes it especially difficult to configure these applications. Even if this first step is completed successfully, changes and failures may disturb those choices and annihilate previous efforts. It is then required to ease these operations by automating them as much as possible. Those observations are at the origin of the development of autonomic computing [1]. Its objective is to let the applications diagnose themselves the problems they are facing and solve them without any external intervention. Of course, issues to be addressed are numerous before reaching a satisfying solution. Here, we focus on one of them: the necessary system support for the development of such autonomic applications.

It is possible to identify several features that must be provided by a platform enabling the construction of autonomic applications. The first and most important one deals with the flexibility of the applications. It is useless to imagine an application being autonomic if it cannot be modified and reconfigured dynamically. Numerous kinds of reconfiguration may be considered, and all of them will have to be supported: from the simple parameterization, to the addition of non-functional properties likely to drastically change the behavior of the application. Besides, the application itself is often the best candidate to collect the measurements that allow the analyze of its own behavior. The platform must then provide the required mechanisms to disseminate these measurements. It must also ease the interactions among the different components of the system: in particular, the platform should be reflexive [2, 3] to give access to the application current state and observed measurements.

This flexibility that we require should not hinder the performance. This, however, has not been the approach chosen in current systems where only one of those properties is developed, but not both. Thus, in the case of platforms specifically tailored for the development of autonomic applications (e.g. AutoPilot [4] or AutoMate [5]), application flexibility (and hence reconfiguration possibilities) has not been emphasized and is insufficiently developed to address the diversity of application needs that we anticipate. At the opposite, in the domains of aspect programming [6] or component systems [7], a high flexibility is provided to the applications built but overheads are introduced that greatly impact the achieved level of performance.

The approach we propose to address this problem builds on an original compromise. We put forward an alternative programming model instead of imposing the use of interpreted language or sacrificing the provided flexibility. This model consists in the stacking of independent components that communicate by exchanging messages. This approach, which — to the best of our knowledge — has never been applied in this context, is not new and several projects have experimented with it, emphasizing its usefulness and expressiveness. Among the first works in this domain are x-Kernel [8] and Ficus [9] (an operating system and a framework for building file systems, respectively). More recently, two new architecture have been proposed: a modular router, Click [10], and SEDA [11] that allows to build efficient Internet services. While most legacy systems provides flexibility through low-level instrumentation (typically at the level of procedure calls), this programming model allows the definition of a custom intervention degree through the choice of component granularity. The platform presented, Pandora, builds upon these techniques and provides a reflexive interface that allows individual components and external applications to dynamically reconfigure the entire system.

We are now going to present (Section 2) some related work pertaining to this study. We describe next the architecture model proposed by Pandora (Section 3). Deployment, execution and control of the applications built on top of it are presented next (Section 4). Next, the platform implementation and some examples of use of the prototype are described (Section 5). An evaluation of the performance of Pandora (Section 6) and a few concluding remarks (Section 7) finish this paper.

2 Related Work

As mentioned before, there exist some platforms that pursue objectives comparable to ours. We are going to detail their characteristics in a first step. Then, we are going to look at software engineering techniques that relates to the approach we have followed for the design of Pandora. Specifically, we will consider component systems and aspect-oriented programming.

2.1 Autonomic Application Platforms

Platforms explicitly dedicated to the construction of autonomic applications are very few. The originality of AutoPilot [4] comes from its work on the definition of sensors and the means to access them efficiently. Those same sensors are used "back way" (to write a value, instead of reading it) to modify the application parameterization. This is however the only reconfiguration feature provided by the platform. Accord [12] and its predecessor AutoMate [5] target applications deployed on grid computing systems. Both were built on top of DIOS [13] for the construction of objects provided with sensors and actuators that allow them to be parameterized dynamically. The main originality of this platform comes from the specification of a language and of a rule execution engine that allow reconfiguration triggering in response to captured events. Unity [14] also targets grid applications. It promotes the use of "autonomic elements" and provides a platform designed to help these elements interact with each others and their environment. In its current state, however, monitoring facilities are rather limited (polling values from OGSA [15] compatible services) and reconfiguration is not supported by the platform itself (it is left to each element to devise their own strategy).

This rapid panorama of autonomic computing platform shows the limits of the reconfiguration mechanisms provided to applications. Dynamic parameterization consists, at best, in choosing between alternative implementations among a set predefined functionalities. Extensibility and modification of non-functional properties are barely addressed.

2.2 Software Engineering Techniques

This limited flexibility of existing platforms leads us to consider the approaches currently in use to address this issue. Two kinds of techniques are clearly put forward: component systems and aspect-oriented programming.

Component Systems. Legacy component systems — .NET, CCM (Corba Component Model) and EJB (Enterprise Java Beans) — are actually rather poorly adapted to the design of flexible architectures. Coarse component granularity, static component binding, and the limited predefined set of non-functional properties provided by the component containers contribute largely to this fact. This has lead to the development of more lightweight systems, with higher performance. Thus, the OpenCOM [7] platform (that builds on the COM component model) allows for both component and bindings dynamic reconfiguration. This is made possible by a fully reflexive interface that enables the platform to access the entire state of the system (the component graph).

The approach followed by Gravity [16] is more original. Each component publishes a list of interfaces it provides and a list of those it requires. As components get in and out of the system, bindings are dynamically established so that all dependencies are satisfied. For our purposes, the main limitation of this technique is the lack of support for "simple" reconfigurations, like the modification of a single parameter value.

Aspect-Oriented Programming. Aspect-oriented programming [6] promotes separation of concerns. Cross-cutting functionalities that are common to several modules of the same program are isolated (those are the *aspects*) and "weaved" with the rest of the application at compile-time or at run-time. The flexibility of these architectures comes from the relative independence between the various entities involved (modules and aspects). It is then possible to modify any of them without disturbing the other elements of the program.

The JAC [17] (Java Aspect Components) platform is the one that is closer to our objectives. In this case, aspects are encapsulated inside components and weaved at run-time (which allows dynamic reconfiguration). These components are also statically configurable to adapt diverse environments. The main limitation of this platform is the lack of support for the reconfiguration of the modules themselves (those whose functionalities cannot be seen as aspects).

3 Architecture Model for Autonomic Applications

Pandora proposes an original architecture model to build autonomic applications. This model builds upon the notion of component and event-based communication to provide the flexibility and the adaptability required by such applications.

3.1 Fine-Grained Independent Components

Autonomic applications are usually considered to be made of a set of relatively independent modules [18]. Each module is supposed to be able to configure itself, detect problems when they occur, and — ideally — solve them. Naturally, all these decisions depend on the state of the system as a whole and modules cooperate with and monitor each other. In the most recent platforms [12, 14] modules are the smallest entities manipulated by the system.

We believe, however, that there are advantages in considering a finer subdivision of modules. In each module, both "business" logic and "autonomic" logic are present. These two aspects are very different by nature and should be clearly separated from each other. Moreover, a large part of the autonomic logic is generic (threshold triggered alarms, rule processing engine, etc.) and could be reused in different modules. Even business code benefits from being further divided into smaller, cleanly bounded, entities. Having an intermediate granularity (coarser than raw instructions and finer than modules), such entities are much easier to monitor. With adequate support from the platform, the programmer is able to easily indicate what are the meaningful variables to monitor and which are the means to modify the component behavior. Besides, parts of the business code consists in the implementation of non-functional properties (the cross-cutting concerns identified by the aspect-oriented programming community) that are essentially reusable from one module to another.

These considerations have led us to propose an architecture model for autonomic applications based on three layers:

1. *components*: components are the basic and self-contained building blocks in the system. Functional and non-functional business code, as well as autonomic machinery, are encapsulated within these objects.
2. *stacks*: components are assembled to form stacks. A stack defines the nature of components to be used and the order in which they are chained. This corresponds to the notion of modules we have mentioned previously.
3. *tasks*: cooperating stacks form a task, which matches the notion of an application, made of several cooperating modules.

Interactions between these three elements are summarized in Figure 1.

Besides, each component may specify a set of parameters, that we name *options*, identified by their name and whose value can be configured at run-time to adapt the behavior of the component. These options may be of different types (numerical, boolean and character strings). However, we cannot represent every parameter type with such basic types (e.g. file handles, set of values, etc.). This is why the model makes it possible to associate a specific pre-processing step before the parameter is given to the component (e.g. transforming a literal host

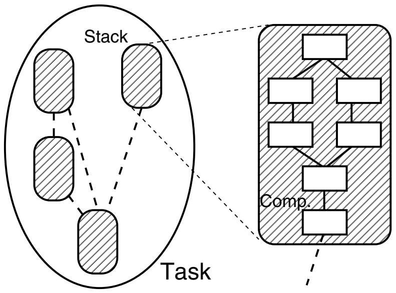


Fig. 1. Interactions between Pandora's elements. Tasks (applications) are made of loosely-coupled stacks (modules). Stacks are made of tightly-coupled components

name into a numerical IP address). Having this step performed outside of the component avoids paying the overhead of these transformations each time a new component instance is created.

The entire independence of the components is an important characteristic of the model, and it is worth considering its implications. A component ignores totally the context in which it is going to be used and interactions with other components are made anonymously. Thus, the reusability of the component is strongly guaranteed. However, the main benefit of this aspect of the model for autonomic applications is simply the very absence of (explicit or hidden) interdependencies between components. This helps simplifying the traditionally hard problem of the dynamic reconfiguration of the application. In this case, each component may be safely adapted and modified without having to fear breaking other components that would depend on him. It must be noted that the problem is simplified, not solved, as the component continues to interact with others and radically modifying its behavior might still impact its peers.

The specification of the chaining of components within stacks, together with their initial parameterization, is made through a dedicated language. In order to ease its usage and comprehension by the platform end-users, a compact stack representation has been preferred. However, it would have been possible to use graph description languages (such as `dot` [19]) or markup languages (such as XML) to achieve similar results. It is beyond the scope of this paper to explain it in details, a full description is available in a previous work [20].

3.2 Event-Based Interactions

Components need to interact with each other. In most legacy component models, communication is performed through direct procedure invocations. This establishes a two-way communication channel: the caller chooses the actual method to call and its arguments and, in the other direction, the callee chooses the value to return. However, our component model uses one-way event-passing communication.

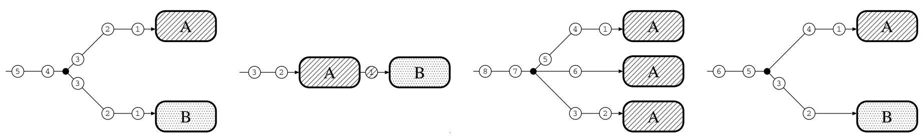
Following this approach offers several advantages. The first one is its conceptual simplicity, which contributes largely to the reduction of the component complexity. A component needs only to define a single interface, the one used to receive an event. This also contributes to the flexibility, the extensibility and the modularity of the platform, which is a primary concern when dealing with autonomic applications. Components may be easily inserted within an established event flow, either to modify it, or, at the opposite to provide, non-functional properties to the whole stack. This could not be easily achieved with an interface-based design. In the latter case, such a generic component would need to implement a large set of interfaces to be composed with all other existing components. When components are added or removed, generic ones should be modified.

By favoring the independence of the components, this communication model is complimentary to the component model we have chosen and helps reinforcing its objectives of flexibility and reusability.

Event forwarding between components is one-way, synchronous, and operates in continuous flow. This means that two components, once the communication is established, are durably connected to each other. At the opposite, communications between stacks, seen as a whole, are asynchronous: events are buffered and consumed whenever the receiving stack decides to do so. It must be noted that a stack, in itself, does not "communicate" with another stack. Rather it is a component that chooses to send an event to a stack, rather than to a component. Similarly, an event sent to a stack is actually processed by a specific component in that stack. Having both communication modes available lets the developer freely choose the best compromise for its application. The number of stacks in the system is not limited and it is perfectly legal to encapsulate a single component within a stack. Synchronous communications are much more efficient (in terms of performance) than asynchronous ones. This is easily explained by the fact that asynchronous communications provide thread-safe buffering support, while synchronous ones obviously do not. Another parameter to take into consideration when choosing between these two modes is that inter-component communications are anonymous, while inter-stack ones are named. Being entirely stand-alone, a component cannot choose which components to communicate with. This is entirely determined in the stack configuration. In the general case, a component willing to transmit an event transmits it to its successor, without knowing its identity. At the opposite, stacks are named (different instances of the same stack are identified by a unique handle or an explicit alias), and a component chooses the name of the stack it wants to communicate with.

Each component has a single input port and an arbitrary number of output ports. The usual case for a component is to have one output port. For those with several output ports, two (mutually exclusive) possibilities exist:

1. *switch*: switch ports are identified by a rank number and it is possible to configure the stack so that components of different nature correspond to each port. This matches roughly the *switch* statement found in most imperative



(a) parallel (b) serial (c) demux (d) switch

Fig. 2. Representation of the different component connections modes. Event are represented by a circle and uniquely identified by a rank number (events with the same number are then identical). In parallel mode, each event is sent to all components. In serial mode, components are chained linearly, events sequentially flows from one component to the next one. In demux mode, events are demultiplexed and sent to the component instance that handles the category they belong to. In switch mode, events are sent to one of several alternative components

programming languages. The choice of the branch in which to forward a specific event is determined by the component itself according to its internal logic.
2. *demultiplexing*: demultiplexing ports allow a component to classify events. As soon as a new category is identified, a new port is dynamically created, while events belonging to an existing category are forwarded to the port with which it had been previously associated. All demultiplexing branches are of the same nature (components of the same type are found in the same order) but made of distinct component instances.

This leads to a wide range of possible component connections when constructing a stack which are summarized and illustrated in Figure 2.

In modes that create multiple branches (i.e. all but *serial*), the actual length and definition of the branches is specified in the stack definition. Events flowing out of related branches (that is, those stemming from the same branch point) are naturally merged in the input of the next component in the stack.

3.3 Flexibility and Reconfiguration

Systems manipulated by Pandora exhibit two degrees of flexibility: first, in component parameterization and, second, in the chaining of these components. Parameterization flexibility relates to option usage to modify variables at run-time: this is rather classical. However, its impact may be of great importance as it is possible to use options to specify an alternative demultiplexing algorithm or change an output port in a switch component, modifying the behavior of the whole application.

The second degree of flexibility exposed by Pandora lies in the specification of component chaining (the stack definition). When several components providing different implementations of the same functionality are available, this allows to choose the solution that best fits the current environment (algorithms trading CPU utilization for memory utilization are quite common). Moreover, the model authorizes (and even promotes) using non-functional components. Their insertion in the stack does not modify the general behavior of the system but might alter the way further events are processed or induce side-effects. One can mention the examples of components managing the balance of processing stages across several machines, the persistence of events they receive, application monitoring, failure detection, synchronization, etc. This last issue is very close to the approach followed in aspect-oriented programming: by modifying the application (components in the original definition), it is possible to weave some aspects (by inserting specific components in the definition).

4 Deployment, Execution and Control

Pandora's architecture is organized around a micro-kernel in charge of stack execution. This notion of micro-kernel references works in the operating system domain which gave birth to several generations of minimal kernels, like Chorus [21], L4 [22] or, more recently, Think [23]. In these systems, functionalities (or services) take the form of independent servers, and live outside the kernel itself. Similarly, functionalities provided by the kernel of Pandora are as limited as possible: their implementation within components has always been preferred. This is the case, for example, of event demultiplexing, inter-stack and inter-machine communications, access control, event persistence, etc. In consequence, the main attributions of micro-kernel are to manage the deployment of the applications, supervise the execution of the stacks and implement the reflexive interface of the platform.

4.1 Libraries and Resources Management

Pandora components come within standard dynamic libraries. To deploy an application in Pandora consists in deploying the libraries containing all components in the stack, together with the appropriate configuration files. Pandora uses two kinds of configuration files: stack definitions and library descriptions. The first one contains any number of textual stack specifications expressed in Pandora architecture description language. The second one deals with libraries: it lets Pandora know in which library each component is, and the location of each library. This location may refer either to a file in the local file system or be an URL, which allows its loading from a remote location. Having these different schemes to access library code allows to build and maintain organization-wide component repositories without requiring participating nodes to share a single remotely mounted file system. Library description files also contain information to specify inter-library dependencies, which are automatically taken care of by the platform.

The different configuration files (stack and library descriptions) are named "resources". In order to ease their management (many such files may be required to run a single application), Pandora uses "meta-resource" files (or simply resource files) that contain a list of locations (file or URL) of other resources. Each such resource is accompanied by a priority which tells the order in which they should be visited. These resource files are considered as resources themselves. This makes it possible to organize resources hierarchically by inserting locations of other (sub-)resources in a higher level resource file. Then, one can build an entire resource tree whose leaves would be stack and library descriptions while resource descriptions would be its nodes. In this end, this means that it is possible to boot strap a whole system from one URL.

4.2 Execution

Stacks are the base entities considered by the platform regarding application execution. To guarantee the integrity of the system, the platform performs a verification stage before starting the actual execution of a stack. It checks the correctness of the stack definition, together with the compatibility of component bindings according to the event types they declare supported for input and output (respectively).

The execution phase starts with the creation of a thread in which the stack will be executed. Each stack is run in its own thread and, respectively, each thread contains at most one stack. This one-to-one mapping between stacks and threads has two advantages. The first one is conceptual: the notion of thread is entirely abstracted and the programmer does not have to worry about it. If she wants her application executed in concurrent threads, she just needs to put her components in separate stacks. The second advantage comes from the guaranty that all components within a stack are executed in a single thread. There is no need, then, to synchronize accesses to stack scoped resources: they are made sequentially. As the number of stacks in the system is not limited, and because stacks can be connected in an arbitrary way, it is perfectly legal to split a single logical module into multiple stacks to allow intra-module concurrency and asynchrony.

Tasks are not explicitely defined by the programmer but dynamically created and managed by the platform to be the connected components of the undirected graph whose nodes are stacks and edges are communication channels established between stacks. For example as stack A sends its first event to stack B, the tasks of A and B are merged. This notion of task is used by the platform to allow specific communication modes (see below, in the next section, sensors and monitors) and to collect dead stack cycles (in the garbage collection terminology). It is clear from this operational definition of tasks that, like stacks, their number is not limited in the system.

The kernel is also responsible for the management of each component lifecycle. As events are produced and transmitted, next components are created lazily, only when they first appear as the destination of an event. Pandora maintains access counters for each component so that it can terminate every component that has become inaccessible after the breaking of a connection. This mechanism, which builds on the explicit termination of component bindings, is complemented by a timeout-based mechanism which collects components after a configurable (possibly infinite) period of inactivity.

4.3 Introspection and Dynamic Reconfiguration

We have said how much needed was flexibility for autonomic applications. This is expressed both in the ability to tune such applications as finely as possible, but also in the possibility to reconsider choices as the environment in which the applications are executed evolves.

The entire configuration of the platform and its current state are exposed by the micro-kernel through a reflexive interface. Stack definitions, option values, resource lists: every aspect of the system that is configurable in a configuration file is accessible through this interface. Furthermore, for each element, it is possible to choose whether to modify the stored definition or its active representation (dynamically modifying the platform behavior). Stack management is also exposed, so that it is possible to know the list of running stacks or request to start a new one or stop another one.

Among these operations, the dynamic reconfiguration of a running stack must be given special consideration. Contrary to all the others, this manipulation implies modifying running component instances and bindings between them. As Pandora components are considered stateful, by default, the platform must pay attention to avoid removing components if not strictly necessary. Pandora does so by computing the minimal set of transformations needed to go from the old definition to the new one in terms of component additions and removals. After a removal stage, remaining component are re-linked to each other inserting new component instances as required by the definition.

A meta-object protocol makes these various reconfiguration operations accessible from applications external to the platform. Pandora provides an interface for this protocol in several programming languages, including C, C++, Perl and Guile. Guile [24] is an implementation of the Scheme language and may be used to write scripts with all the standard construction of the original language (procedure definition, flow control, etc.) augmented by primitives accessing the kernel reflexive interface. The ability to write such "control scripts" is original to Pandora and eases the rapid prototyping of (partially) autonomic applications.

However, for autonomic applications to analyze and reconfigure themselves, the utilization of the above protocol is not optimal in terms of performance, as it is designed to allow external applications to interact with the platform. When accesses are made from within the autonomic application (the task in Pandora's terminology), much of the overhead required to locate the targeted option in the system and to serialize the results in the response can be avoided. We have then introduced a specific mechanism that allows a component to "publish" values and make them accessible to all other components within the same task. This operation is made through a dedicated object that we have called a *sensor*. Each sensor is given a name and several component instances may update a single sensor. Accesses are made through another object, called a *monitor*. Monitors are initialized with a set of sensor names they are related to and with a function to apply to the values in order to process them. When this function is actually called depends on the access mode that was chosen for the sensors. There exist two different access modes: a passive mode where monitors pull the values from the sensors when they need it, and an active mode where sensors push the values to the monitors as soon as they are modified. Choosing the best appropriate mode depends on the relative read and write frequencies of monitors and sensors, respectively. Finally, an automatic mode is provided that let the platform do this choice according to access counters it maintains.

5 Implementation

A prototype of this architecture has been developed and has been used in several applications. We present them here briefly.

5.1 Prototype

We have developed a software platform that implements the architecture we have described. It represents more than 50 000 lines of C++ code and, besides the kernel, is made of about 100 components. Approximately a third of those are “base” components: these are components that implement non-functional properties and that may be used in any stack.

Pandora has been ported and tested on a large number of systems, including Linux, FreeBSD, NetBSD, Solaris, Digital Unix (Tru64) and, for the kernel and base components only, Win32. The platform is distributed in its most recent version under an open-source license by INRIA (free for non-commercial use) at the following URL: `http://www-sor.inria.fr/projects/relais/pandora/`.

5.2 Applications

The Pandora platform has already been used in several projects [20, 25–27]. However, the application that emphasizes most the flexibility of Pandora and the features it offers to build autonomic applications is C/SPAN [28]. C/SPAN is an autonomic Web proxy cache that builds, for the one part, on C/NN [29], a flexible Web cache, and, for the other part, on a HTTP monitoring stack on top of Pandora. In this system, C/NN and Pandora are in a tight interaction loop: C/NN, according to its environment (disk space, request rate, etc.), tunes the behavior of Pandora using its reflexive interface. Respectively, Pandora reconfigures C/NN according to the traffic patters it observes.

6 Performance Evaluation

In this evaluation of the performance of the platform, we have focused on two specific points: the overhead related to the application slicing into components and that related to introspection operations.

All tests have been performed on the same machine which uses a 2.4 GHz Pentium IV processor, running the version 2.6 of the Linux operating system. The measurements that we present are computed from the average of 50 successive runs. The standard error associated with these averages has never gone over 1%. The various procedure execution time have been measured with a loop that executes each one million times. Total execution time expressed in milliseconds gives then the cost a single iteration expressed in nanoseconds.

6.1 Component Traversal

To evaluate the overhead related to the slicing into components, we have measured the time needed for an event to flow through one component, i.e. the time needed to go from one component to its successor. Results presented in Figure 3 show that this time is about 50 ns. Given the other measurements we have performed, we see that this time is superior to the one needed to make simple library calls, but inferior to the one required to make a floating-point division or a system call. This indicates that for non-trivial applications (those that actually do something between each event transmission), the overhead related to the slicing is rather limited, if not negligible.

6.2 Introspection Primitives

To monitor its own behavior, an autonomic application must permanently watch the sensors it is provided with. At the opposite, reconfigurations are supposed to happen only in exceptional circumstances. Then, the most performance critical operation for those applications is the reading of a sensor value, and this is the one we have chosen to evaluate. For the sake of comparison, we have also measured the time needed to read a standard variable when using the reflexive interfaces of two languages commonly used to build flexible applications: Java and C#. In each language, we have reduced the operation to its most simple expression: reading the value of an integer field of an object instance. In both cases, the code used is a one-liner. For Java, we have used different virtual machines, with and without dynamic compilation (Just In Time). We have also statically

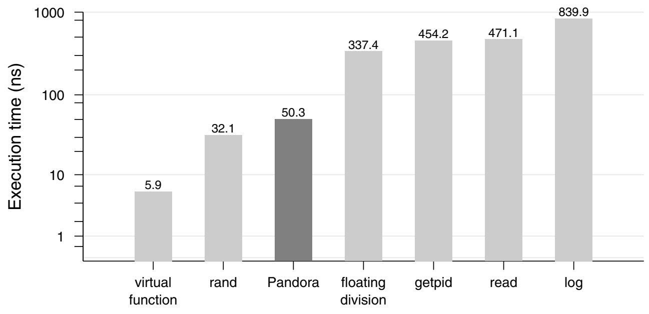


Fig. 3. Average execution time for various library functions, compared to Pandora component traversal time

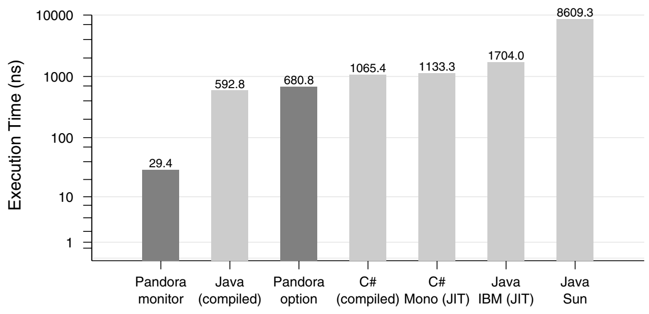


Fig. 4. Average execution time for several language-based introspection functions (Java and C#) compared to those provided by Pandora

compiled the program into native code using the GNU compiler [30]. For C#, we have used the Mono [31] environment whose virtual machine supports both static and dynamic compilation. The results of these experiments are shown Figure 4. Rather than studying the relative performance of compilers and virtual machines compared to each other, what we would like to focus on are orders of magnitude. Pandora sensors (in passive mode) are about 20 times more efficient than natively compiled Java code. The latter, however, corresponds roughly to the time spent when using Pandora options through its external interface. Unsurprisingly, the use of true virtual machine (no static compilation) degrades performance further and the absence of dynamic compilation makes them really catastrophic. This shows how Pandora, for an equivalent CPU load, can support a much higher number (one or two orders of magnitude) of sensors, and thus of applications compared to languages that have been usually used in this domain. It is Pandora specialization in these tasks (the platform has been designed and optimized for this exact purpose), as opposed to the necessary generic approach of a high-level programming language, that explains those differences.

7 Conclusion

We have presented Pandora, a platform for the construction of autonomic applications. Pandora builds on an original programming model, the stacking of components communicating with message exchanges, that provides a compromise between flexibility and performance. The resulting Pandora component model is much simpler than legacy approaches that proceed with standard function calls. The architecture of the system is organized around a micro-kernel that is responsible for managing the system resources (configuration files) whose hierarchical organization eases large-scale deployments. It is also responsible for the chaining of the components according to specified configurations. Its last role is to expose a reflexive interface and propose the necessary abstractions for an application programmer to dynamically configure and reconfigure the entire system. This architecture has been implemented — the prototype is available freely on the Internet — and several applications have already used it. A performance evaluation of the system has shown that its implementation backs up our initial objective to reconcile flexibility and performance.

With Pandora, autonomic application developers needs only to concentrate in implementing the “business logic” of their application. The platform indeed factorizes out most non-functional aspects of the application and provides useful abstractions to deal with monitoring. After some initial effort required to design the application according to Pandora’s programming model, easy and powerful deployment and administration support is provided by the system. More importantly, Pandora also makes the application highly flexible and dynamically reconfigurable, basically for free.

References

1. Kephart, J., Chess, D.: The vision of autonomic computing. Computer Magazine, IEEE (2003)
2. Smith, B.C.: Reflection and semantics in lisp. In: Proceedings of the 11th ACM SIGACT-SIGPLAN symposium on Principles of programming languages, ACM Press (1984) 23–35
3. Maes, P.: Concepts and experiments in computational reflection. ACM SIGPLAN Notices **22** (1987) 147–155
4. Ribler, R.L., Vetter, J.S., Simitci, H., Reed, D.A.: Autopilot: Adaptive control of distributed applications. In: Proceedings of the The Seventh IEEE International Symposium on High Performance Distributed Computing, IEEE Computer Society (1998) 172
5. Agarwal, M., Bhat, V., Li, Z., Liu, H., Khargharia, B., Matossian, V., V.Putty, Schmidt, C., Zhang, G., Hariri, S., Parashar, M.: Automate: Enabling autonomic applications on the grid. In: Proceedings of the Autonomic Computing Workshop (AMS 2003), Seattle, WA (2003)
6. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.M., Irwin, J.: Aspect-oriented programming. In Aksit, M., Matsuoka, S., eds.: ECOOP'97 — The 11th European Conference on Object-Oriented Programming. Volume 1241 of Lecture Notes in Computer Science., Jyväskylä, Finland, Springer-Verlag (1997) 220–242
7. Clarke, M., Blair, G.S., Coulson, G., Parlavantzas, N.: An efficient component model for the construction of adaptive middleware. Lecture Notes in Computer Science **2218** (2001)
8. Hutchinson, N.C., Peterson, L.L.: The *x*-Kernel: An architecture for implementing network protocols. IEEE Transactions on Software Engineering **17** (1991) 64–76
9. Heidemann, J.S.: Stackable layers: An architecture for file system development. Technical Report UCLA-CSD 910056, University of California, Los Angeles, CA (USA) (1991)
10. Kohler, E., Morris, R., Chen, B., Jannotti, J., Kaashoek, M.F.: The click modular router. ACM Transactions on Computer Systems **18** (2000) 263–297
11. Welsh, M., Culler, D.E., Brewer, E.A.: SEDA: An architecture for well-conditioned, scalable internet services. In: 18th Symposium on Operating Systems Principles, Lake Louise, Canada (2001) 230–243
12. Liu, H., Parashar, M., Hariri, S.: A component-based programming model for autonomic applications. In: Proceedings of the 1st International Conference on Autonomic Computing (ICAC 2004), New-York, NY, IEEE Computer Society (2004)
13. Muralidhar, R., Parashar, M.: A Distributed Object Infrastructure for Interaction and Steering. In R. Sakellariou, J. Keane, J.G., Freeman, L., eds.: Proceedings of the 7th International Euro-Par Conference (Euro-Par 2001),Lecture Notes in Computer Science. Volume 2150., Manchester, UK, Springer-Verlag (2001) 67–74
14. Chess, D.M., Segal, A., Whalley, I., White, S.R.: Unity: Experiences with a prototype autonomic computing system. In: Proceedings of the 1st International Conference on Autonomic Computing (ICAC 2004), New-York, NY, IEEE Computer Society (2004)
15. Foster, I., Kesselman, C., Nick, J.M., Tuecke, S.: The physiology of the grid: An open grid services architecture for distributed systems integration. Open Grid Service Infrastructure WG, Global Grid Forum (2002) `http://www.globus.org/research/papers/ogsa.pdf`.

16. Hall, R.S., Cervantes, H.: Gravity: supporting dynamically available services in client-side applications. In: Proceedings of the 9th European software engineering conference held jointly with 10th ACM SIGSOFT international symposium on Foundations of software engineering, ACM Press (2003) 379–382
17. Pawlak, R., Seinturier, L., Duchien, L., Florin, G., Legond-Aubry, F., Martelli, L.: JAC: An aspect-based distributed dynamic framework. Software: Practice and Experience (SPE) (2004)
18. White, S.R., Hanson, J.E., Whalley, I., Chess, D.M., Kephart, J.O.: An architectural approach to autonomic computing. In: Proceedings of the 1st International Conference on Autonomic Computing (ICAC 2004), New-York, NY, IEEE Computer Society (2004)
19. Ellson, J., Gansner, E., Koutsofios, L., North, S.C., Woodhull, G.: Graphviz — open source graph drawing tools. Lecture Notes in Computer Science **2265** (2002)
20. Patarin, S.: Pandora: support pour des services de métrologie à l'échelle d'Internet (english title: Pandora: Support for Internet Scale Monitoring Services). PhD thesis, Université Pierre et Marie Curie – Paris 6 (2003) In French.
21. Rozier, M., Abrossimov, V., Armand, F., Boule, I., Gien, M., Guillemont, M., Herrmann, F., Kaiser, C., Langlois, S., oñard, P.L., Neuhauser, W.: CHORUS distributed operating system. Computing Systems **1** (1988) 305–370
22. Härtig, H., Hohmuth, M., Liedtke, J., Schönberg, S., Wolter, J.: The performance of μ-Kernel-based systems. In: Proceedings of the 16th Symposium on Operating Systems Principles (SOSP-97). Volume 31 of Operating Systems Review., Saint Malo, France, ACM Press (1997) 66–77
23. Fassino, J.P., Stefani, J.B., Lawall, J., Muller, G.: THINK: a software framework for component-based operating system kernels. In: 2002 USENIX Annual Technical Conference, Monterey, CA (2002) 73–86
24. Jaffer, A., Carrette, G., Stachowiak, M., et al.: Guile, project gnu's extension language. software (2002) `http://www.gnu.org/software/guile/`.
25. Patarin, S., Makpangou, M.: On-line Measurement of Web Proxy Cache Efficiency. Research Report RR-4782, INRIA (2003)
26. Fessant, F.L., Patarin, S.: MLdonkey, a Multi-Network Peer-to-Peer File-Sharing Program. Research Report RR-4797, INRIA (2003)
27. Patarin, S., Salamatian, K., Friedman, T.: The Pandora network monitoring platform (2004) Submitted for publication.
28. Ogel, F., Patarin, S., Piumarta, I., Folliot, B.: C/SPAN: a Self-Adapting Web Proxy Cache. In: Proceedings of the Autonomic Computing Workshop (AMS 2003), Seattle, WA (2003)
29. Piumarta, I., Ogel, F., Baillarguet, C., Folliot, B.: Applying the vvm kernel to flexible web caches. In: Proceedings of the IEEE Workshop on Hot Topics in Operating Sy stems, HOTOS-VIII, Schloss Elmau, Germany (2001) 155
30. Bothner, P., Haley, A., Levy, W., et al.: The gnu compiler for the java programming language. software (2004) `http://gcc.gnu.org/java/`.
31. de Icaza, M., Molaro, P., Pratap, R., Porter, D., et al.: The mono project. software (2004) `http://www.go-mono.com/`.