

Toward Deeply Adaptive Societies of Digital Systems

Antonio Carzaniga[†] Giovanni Denaro[‡] Mauro Pezze^{†,‡} Jacky Estublier[§] Alexander L. Wolf^{*}
antonio.carzaniga@unisi.ch denaro@disco.unimib.it mauro.pezze@unisi.ch jacky.estublier@imag.fr a.wolf@imperial.ac.uk

[†]University of Lugano
Lugano, Switzerland

[‡]University of Milano-Bicocca
Milano, Italy

[§]Lab. Log., Sys. et Reseaux
Grenoble, France

^{*}Imperial College London
London, United Kingdom

Abstract

Modern societies are pervaded by computerized, heterogeneous devices designed for specific purposes, but also more and more often capable of interacting with other devices for entirely different purposes. For example, a cell phone could be used to purchase a train ticket on-line that could later be printed by a vending machine at the train station. This type of open environment is what we call a society of digital systems. In this paper, we outline the characteristics of societies of digital systems, and argue that they call for a new approach to cope with unforeseen interactions, possible incompatibilities, failures, and emergent behaviors. We argue that designers can not assume a closed or homogeneous world, and must instead naturally accommodate dynamic adaptations. Furthermore, self-adaptability, that is, the ability to adapt autonomically to a changing environment, also poses problems, as different adaptation strategies may interfere negatively, leading to unstable behaviors. As an initial concrete contribution to solve this problem, we propose a method to support the graceful integration of devices and software systems in an open environment. The method uses management information, and is specifically centered on the idea of expressing self-adaptation operations as change sets over the management information base.

1 Societies of digital systems

Today, the use of digital systems pervades all areas of our lives. Applications are for individuals, from domestic to automotive, from personal communication to entertainment, as well as for small and large communities, including health, environment monitoring and management, transportation, and energy production and conservation. Not only digital devices are ever more diverse and pervasive, they are also often capable of interacting with each other to integrate their services. For example, it is now considered normal for a cell phone to connect to the audio system of

an automobile. Closer to the traditional computing world, it is also considered normal for an idle computer to offer its computational resources to applications running elsewhere on the network, and of course, many more such interactions are technologically feasible today.

We refer to such assemblies of interacting systems and devices as *societies of digital systems*. Some of these societies exist and evolve around and within human societies, while others are made of pure software systems. Nevertheless, in both cases we use the analogy of a society, not to tie the digital systems to their human context, but rather to emphasize the open nature of the environment in which the digital devices and systems operate as well as the open nature of the cooperation between systems. This paper discusses some implications of this openness on the engineering of the software for such societies.

There are several general challenges in engineering software for open collaborations and open environments. One is to manage or at least tolerate incompatibilities in the interfaces of other systems or other deviations from the expected behavior of other systems. Another problem is the need to adapt to a dynamic and unpredictable environment where failures are normal and resources (e.g., network connectivity) may be sporadically available or may undergo frequent configuration changes [4, 6].

Beyond the individual systems, the increased complexity of societies of digital systems reveals side effects and unexpected interactions that induce unforeseen global and local behaviors. These collective behaviors result from the interoperation of several devices and systems, and are therefore difficult to model, to understand, and to reproduce in testing and debugging environments. In general, these behaviors may also be difficult to control, and in some cases may even be undetectable by each individual system.

Our general hypothesis is that software systems must include appropriate *adaptation* procedures to autonomically respond to the needs of open environments and open interactions. The contributions we intend to make with this paper are ideas on how to support and engineer those adaptation procedures. We first formulate a high-level notional

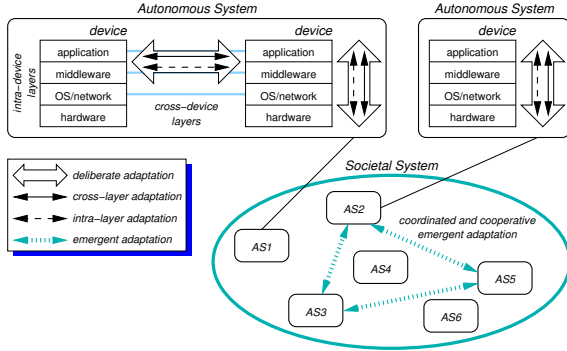


Figure 1. Deeply adaptive systems.

model of societies of digital devices, and specifically of their ability to adapt both as individuals and collectively. We refer to this process as *deep adaptability*. Then, based on this model we propose the foundational elements of a concrete infrastructure to support deep adaptability.

2 Deep adaptability

At a high-level, we approach the problem of supporting open interactions by conceiving of the problem of self-adaptability holistically. We argue that, by doing that, we can provide systems with a substantially and significantly improved ability to manage themselves, whether to accommodate changes, respond to failures and attacks, or to take advantage of new resources. We refer to this approach as *deep adaptability* because it fundamentally integrates adaptation mechanisms across different devices, systems and architectural layers.

To make the discussion more concrete and tractable, we introduce terminology and assumptions that help structure the problem. In our conception, depicted in Figure 1, a device is a platform for hardware and software components, within which we can recognize layers. The layers within a device are intra-device layers. A system that spans multiple devices has corresponding cross-device layers. Adaptation can involve either a single layer, intra-layer adaptation, or multiple layers, cross-layer adaptation.

The holistic view requires us to rethink the concept of *system*. At its most basic, a computer-based system is any assembly of hardware and software components interacting to produce some effect. We see systems arising in two, very different ways: first through a deliberate design and assembly process, and second through a circumstantial (*ad hoc*) and emergent interaction of components. Systems of the first type operate under administrative control, while those of the second type operate in an apparently non deterministic regime that resembles anarchy. Each exhibits characteristics that engender fundamentally different require-

ments for realizing adaptability: A self-adaptive deliberate assembly of components is an autonomous system admitting to some form of deliberate adaptation under some form of administrative control [5]. A self-adaptive society system is a (potentially anarchic) collection of interacting autonomous systems. A societal system exhibits emergent adaptation, since the adaptation emerges from a cooperative behavior among autonomous systems. The society of digital systems is the highest conceivable level of system abstraction, incorporating the full range of adaptability challenges, from well-understood and well-supported intra-device/intra-level adaptations on the one hand, to the unexplored cross-device/cross-layer adaptations necessary for deep adaptability on the other.

3 Sample scenarios

To make the discussion more concrete, this section describes two scenarios in which deep adaptation plays a significant role, and in which it may also lead to conflicts.

In the first scenario we consider a data center hosting the server side of a large-scale web application, such as a popular game distributed within a social network, or a content streaming service. Such web applications are typically hosted within an application server that automatically manages the allocation of computational resources to the various application components. Moreover, within the data center, the application servers are themselves replicated over several hosts.

Considering a typical three-tier architecture, we assume for simplicity that each application consists of one stateless component that can run as a single thread within the application server. The application server has a thread pool that it allocates according to the dynamic load of each application. As more client requests for application A arrive, the application server allocates more threads to run multiple instances of component A . Outside of the application server, another load-balancing mechanism is implemented across the entire data center. Such load balancers are typically implemented by replicating service components (and possibly persistent data items) over different hosts, and by directing each request to one of the corresponding application server.

We consider a situation in which these two adaptation mechanisms incur a conflict. We start from a deployment where application A runs only on host H_1 with T_A threads allocated to it. At this point, a surge in requests for application A leads the application server on H_1 to double the allocation of threads to application A , to $2T_A$, shifting those threads from another application, B , running on that application server. At the same time, the global load balancer responds to the surge of requests for application A by activating a replica of A with T_A threads running on a second host H_2 . Notice that, in this case, the local and global adapta-

tion mechanisms are not necessarily conflicting, since they both increase the resources allocated to application *A*. Yet, the combination of both mechanisms, each one designed to double the resources allocated to *A*, results in a triple allocation. This overreaction may in turn cause the two mechanism to react again, this time in the opposite direction, only to fall into a cycle of continuous oscillations.

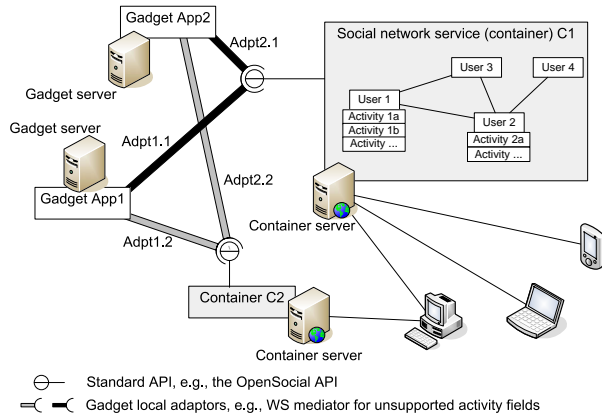


Figure 2. Social Networking Scenario

We now discuss a second scenario where self-adaptation addresses functionality rather than performance. We consider a set of applications that operate within a social network: a service (or *container*) that manages the social relationships in the network, and a set of programs (or *gadgets*) that provide user oriented functionality over the network. These applications are loosely coupled and administratively autonomous, and interact through web services exploiting standard APIs such as Google OpenSocial.¹ This scenario is depicted in Figure 2.

Since it is desirable that gadgets interact with several containers (e.g., Facebook or MySpace), we consider a self-adaptive mechanism that automatically checks client- and server-side incompatibilities, and that deploys ad-hoc *mediators* to guarantee interoperability. For instance, we consider a gadget that publishes information on the activities of its users using OpenSocial activity records. These records consist of a set of typed fields, such as *TITLE*, *BODY* and *URL*, but the API mandates only the handling of *TITLE*, while other fields may be ignored by the containers. To guarantee functional correctness, the self-adaptive mechanism detects whether some field used by the gadget are not implemented by the target container, and accordingly deploys mediators that use a valid field (e.g., *TITLE*) to store data of ignored fields, and consistently restore the correct contents of the fields when retrieving the activities.

This adaptation mechanism works well for isolated gadget-container pairs [2], but may incur interference and

conflicts when different instances are used independently by a set of gadgets. With reference to the *OpenSocial* API, we consider for example a gadget that monitors Web navigation activities: when a user visits a Web site, the gadget records the title and the URL into corresponding activity fields, and on-demand visualizes the navigation history. If the container does not implement the *URL* field, the adaptation mechanism may embed URLs in the *TITLE* text, using a specific tag for mark-up. At the same time, another gadget may want to use the URLs of an activity, but may be configured to recognize another mark-up within the *TITLE* text. When the two gadgets work with activities shared by users on a container, the two adaptation mechanisms may interfere with a loss of functionality.

4 An infrastructure for deep adaptability

Our long-term vision is one where software systems are almost completely adaptive and self-managed. This vision is obviously very ambitious in scope. In fact, the main difficulty of realizing this vision lies in its broad generalization of the use of self-adaptive mechanisms to provide self-management for an almost completely open environment. We can imagine three general strategies to cope with this difficulty. One is to design and implement a universal automatic controller applicable to all systems and all types of adaptation and management tasks [1]. However, we consider this approach unrealistic for complex digital societies.

A second and more realistic approach is to design specific controllers for specific classes of applications or groups of applications [5]. This approach, while perhaps practical in some cases, also suffers from a limited scalability. This is because many ad-hoc controllers in large groups of systems are likely to generate more conflicts and incompatibilities than the ones they are supposed to solve. In fact, this approach does not change the nature or the architecture of systems and societies in a substantial way.

The third approach tries to combine at least part of the benefits of a universal controller with the reality of a diverse and open environment. The general idea is to provide a common infrastructure upon which specific controllers can be realized, and through which they can cooperate [3]. This is the approach we take and explore in detail here.

Concretely, our proposal is based on the use of a uniform representation of application-management information and adaptation plans. At a high-level, the idea is to express self-management adaptations (or any other dynamic evolution of a system) as *change sets* over the application-management information base. This denotation of adaptations reduces the detection of conflicting adaptations to the detection of a conflict between two or more change sets. Once two or more systems detect a conflict in their adaptation strategies, they can proceed with a conflict-resolution protocol.

¹<http://code.google.com/apis/opensocial>

This idea is enabled by two existing and relatively mature technologies. The first is application management. Application management refers to a number of techniques to monitor and manage running applications or systems. The most relevant element of application management that we propose to leverage in this case is the so-called *management information base*, which defines, stores, and provides access to a description of the state of an application. Such descriptions conform to detailed and standardized models of applications, systems, and their resources, and have well-defined external representations amenable to the basic text processing required to represent compact change sets.²

The second enabling technology is content-based publish/subscribe communication. This communication style allows applications to simply publish their adaptation plans (again, in the form of a change set) without knowing which other applications might need to be informed of those plans. This is because each application subscribes to any adaptation plans that refer to managed resources that affect the application's behavior. So, when an adaptation plan is published, the content-based communication system delivers the plan to all the relevant subscribers.³

With uniform knowledge of all the relevant adaptation plans, each application can merge all the plans, and therefore detect adaptation conflicts. This process is local, is based on a purely syntactic merge operation, and is therefore very efficient. Furthermore, every application involved in a conflicting set of adaptation operations would receive all the relevant change sets, and therefore would consistently detect the same conflict.

Once a conflict is detected, applications could react by executing a deterministic conflict-resolution procedure without generating or requiring additional interactions between the applications. For example, all applications except the one with the highest identifier could simply decide to abort their adaptations. Of course, other more sophisticated procedures are also possible. However, in this paper we do not concern ourselves with the specific design and implementation of conflict-resolution procedures. We only note that these procedures can be seen once again as merge operations, and may be context-dependent, may require explicit interactions and negotiations between applications, and may be deterministic or non-deterministic.

Change sets that define adaptation plans have a time dimension, too, in the sense that they also specify a scheduled time and duration for the application of the plan. The time dimension could allow distinct adaptation mechanism that operate at two different levels to coexist constructively. For example, one adaptation mechanism could deal with short-term "tactical" adaptations, while another one could implement a long-term "strategic" adaptation. because of the dif-

ferent schedules, the two mechanism may or may not incur conflicts. In case they do, the time dimension can be used to resolve the conflict with the right priority. However, even if the two plans do not conflict, the knowledge of the long-term plan may be useful in guiding the formulation of the short-term plans.

5 Conclusions

In this paper, we propose a change of perspective for developing and maintaining large societies of digital systems. The new perspective departs from classic self-adaptation approaches that focus on adaptation strategies for a single application, toward a holistic view that focuses on adaptation for a large population of applications and devices operating in an open environment. Concretely, we propose to establish a common infrastructure to support coherent adaptation among independent systems as well as across the levels of an individual system. The primary service of this infrastructure consists of exporting the state of systems (i.e., their management information) as well as the plans to change that state (i.e., their self-adaptation strategies) in a uniform way. The mechanism we propose relies on mature technologies, admits efficient procedures for the detection of conflicts, and supports reconciliation over short-term and long-term adaptations.

References

- [1] R. J. Anthony. Policy-centric integration and dynamic composition of autonomic computing techniques. In *Proceedings of the Fourth International Conference on Autonomic Computing*. IEEE Computer Society, 2007.
- [2] G. Denaro, M. Pezzè, and D. Tosi. Towards autonomic service-oriented applications. *International Journal of Autonomic Computing*, 2009 to appear.
- [3] C. Gacek, H. Giese, and E. Hadar. Friends or foes?: a conceptual analysis of self-adaptation and its change management. In *Proceedings of the 2008 international workshop on Software engineering for adaptive and self-managing systems*. ACM, 2008.
- [4] J. Kramer and J. Magee. Self-managed systems: an architectural challenge. In *2007 Future of Software Engineering*, pages 259–268. IEEE Computer Society, 2007.
- [5] P. Oreizy, N. Medvidovic, and R. N. Taylor. Runtime software adaptation: framework, approaches, and styles. In *Companion of the 30th international conference on Software engineering*, pages 899–910. ACM, 2008.
- [6] G. Valetto and G. Kaiser. Using process technology to control and coordinate software adaptation. In *Proceedings of the 25th international conference on Software engineering*, pages 262–272. IEEE Computer Society, 2003.

²E.g., for Java applications see the JMX framework.

³E.g., for Java applications see the JMS framework.