Lightweight Services for Embedded Systems

Nikola Milanovic, Jan Richling, Miroslaw Malek
Humboldt Universität, Berlin
{milanovi,richling,malek}@informatik.hu-berlin.de

Abstract

We propose creating service oriented-architecture for embedded systems. We introduce the concept of contracts for embedded systems, and propose architectural and design solutions for accessing embedded systems using lightweight services based on contracts. We define proxy, translator and full service architectures, that enable us to link with a variety of classes of embedded devices. By treating embedded systems as services we expect to make a concept of self-descriptive and reusable embedded systems a step closer to reality.

Indexed terms: embedded systems, services, contracts

1. Introduction

The paradigms of computer science are shifting and we can identify two major groups: very big and very small systems. In one group are mainly servers and server farms, and in the other are mainly embedded systems. Today, there is very little or no communication between them at all.

The problem is not only in the communication infrastructure, because the current trends of convergence of computing and communication are making embedded systems capable of communicating using different wired and/or wireless technologies. The problem is how we design and implement embedded systems. While we try to keep big systems as open as possible (since it is their primary role), we tend to seclude and isolate embedded systems without providing easy ways to add a custom interface to them. Embedded systems are still mainly seen as vendor-specific and task-oriented products, and not as components that can be easily manipulated and reused.

Our goal is to explore different architectural approaches that can be used to make embedded systems more accessible. We see embedded systems as components that provide services. Therefore, we need ways how to describe the services they offer, how to discover and to use them. Our vision is one of self-descriptive, reusable and accessible off-the-shelf embedded systems.

2. Related Work

Service-oriented computing is a computing paradigm that uses services as basic building blocks for application development. Services are self-describing and open components that support rapid and seamless access and integration. Services are offered by service providers and they are used by service consumers. Therefore, the main architectural units in service-oriented computing are service description, service discovery and service consummation.

There are many service architectures that are proposed today. One of the most prominent is Web Service Architecture [8]. Web Service is identified by a URI, and described using Web Service Description Language (WSDL). Interaction between Web Services is typically performed with Simple Object Access Protocol (SOAP) calls that carry XML information. Service discovery is performed using Universal Description, Discovery and Integration (UDDI), which is a protocol for directories that contain Web Service descriptions. Service consumer and service directory are not interested how a service is implemented. It is enough to have WSDL description since it contains all necessary information how to invoke a service and receive results. Service implementation can even change without notifying the client.

There are other approaches to service oriented computing, like JINI services [6], Open Grid Service Architecture (OGSA) [3], or OSGi Service Platform [2]. They are all targeted to specific usage domains, but retain the basic idea of providing service description, discovery and consummation. We will try to show how this approach can be used for providing service oriented architecture for embedded systems. We will assume a model similar to Web Service architecture, and try to explore what kind of description we need for embedded systems and how to modify the way we design

embedded systems in order to be able to access and use them in a service-like architecture.

The rest of the paper is structured as follows. In Section 3 we introduce a concept of contracts. In Section 4 we show how to create contracts for embedded systems, and in Section 5 we explore architectural solution for accessing embedded systems using contracts via lightweight services. Section 6 concludes with pointers for future work in this area.

3. Specification Contracts

The first task that we have to solve, if we want to provide a service architecture for embedded systems, is to propose a way for uniform specification of systems that would constitute such architecture. It is a great challenge to come up with a specification scheme that is both general and lightweight. We are aware that it is impossible to provide a meaningful interface specification of an open component without considering the context-of-use in a particular application environment [4]. Therefore, we will mark the issues that are conceptually important when trying to specify a component, be it a software component, a web server, or an embedded system. When we cover basic issues we will show the peculiarities of embedded systems specification.

We introduce the concept of component contract. Contract is coupled with a component and tells the client of that component what conditions the component requires for correct execution, and what it will deliver if those conditions are met. In theory, contract should be enough for a client to decide how to use a component. But knowing how to use a component is only one thing. A client should be aware what and how a component will deliver, too.

There are several important reasons why contracts make such a good prospect to have when building a service-oriented architecture:

- Equipping components with contracts facilitates reuse and makes it much safer.
- Contracts can help in comparing and choosing between similar components.
- Contracts fit perfectly as semantically extended service descriptions, which allows us to treat components as services.
- Adding composition behavior to contracts can help with automated component composition.

There is an elaborate effort to introduce contracts into modern software engineering, and it is called Design by Contract [5]. Contracts are attributed to software components, such as .NET assemblies or Enterprise Java Beans. However, this idea has the root in the Eiffel programming language, which makes the use of contracts mandatory. We cannot deploy an Eiffel library without specifying basic attributes, such as preconditions, postconditions and invariants. Preconditions are obligations of the caller, or consumer of a component. They state what the caller must ensure in order that component executes correctly. Postconditions, on the other side, are the obligations of a component. They state what the component 'promises' to deliver if preconditions are ensured. Invariants describe static properties of a component, that must be preserved before and after each use of a component.

One way of extending this idea is to include non-functional properties in contracts. We have already done some work in this area [7] and extended a contract for software components with non-functional properties such as security, dependability, performance, and rendering. In the next section we will describe how to come up with a contract scheme for embedded systems.

4. Contracts for Embedded Systems

As stated in Section 1, our goal is to apply the concepts from Section 3 in embedded systems to develop a service oriented architecture that allows simple and seamless access to nearly all kinds of computer systems. In order to do this, we have first to see what embedded systems are and how one embedded system may differ from another. Second, we have to see which properties are important and have to be part of such contracts.

4.1. Devices

One of the major problems of developing contracts for embedded systems is the fact that "embedded system" does not cover just one concept or class of devices. Instead, "embedded systems" means a whole range of devices from very small low power singles solutions up to large multiprocessor systems. Therefore, we start here with a short overview of embedded systems and introduce the types of them that we distinguish.

Common to all embedded systems is the property of "being embedded". This means the computer system is not just used to control some technical system, it is part of that system and is, in most cases, connected to it throughout the entire lifetime. Compared to commercial-off-the-shelve (COTS) computers, embedded systems are typically tailored to a given application and are limited in resources such as CPU cycles, storage, power and software.

This also applies to interconnectivity issues and implies that it is not possible to assume that each embedded system is able to communicate using some standard communication protocols because in general this is not needed to fulfill the needs of the application.

Therefore, new communication schemes such as service oriented architectures have to obey that fact and should not try to force the usage of a specific high-level protocol for each device. But, on the other hand, such an architecture would make no sense without the ability to interact "somehow" with all kinds of devices.

In order to solve this problem, we first look onto several types of embedded systems. This is intended to further motivate the statement given above and to show paths to a solution based on contracts and services.

We are interested in communication, therefore we use the abilities of communication (with respect to a service architecture) as criteria to classify embedded systems:

4.1.1. No communication In very limited application domains devices are used that only interact with their physical environment and have no possibility to exchange information with other devices[1]. Such a device is completely isolated and cannot be included into any kind of communication architecture.

An example is a pacemaker: It gets installed, gets information from sensors about heartbeat, blood pressure and whatever (environment), and based on that it performs its function. There is no way[2] to exchange signals with a pacemaker, and it would be very dangerous to do so indeed.

Adding the missing functions regarding service architecture to such a device would dramatically increase costs because in most cases the requirements to fulfill the function are much lower than these to implement the missing communication abilities.

4.1.2. Proprietary physical communication The next type of devices (which are the majority today) are systems that are able to interchange information using proprietary[3] methods both at physical and logical layer. This ability does not necessary mean that the device is "networked", it is sufficient that it is able to deliver data to other systems.

Examples here are control systems in cars for, e.g., airbags, engine, comfort functions or the braking system.

Adding the missing functions regarding service architecture has the same problems as discussed above.

4.1.3. Proprietary logical communication With increasing complexity devices may use standard communication techniques at the physical level (e.g., ethernet) or both at physical and transport level (e.g., Ethernet and IP) and a proprietary protocol above that to interchange data with other systems that use the same technology.

Example here is, e.g., the remote control of some cameras using ethernet links or IP enabled applications using proprietary application protocols.

The overhead of adding the missing functions regarding service architecture depends on the application but may again need more resources than the application itself.

4.1.4. Full communication At the topmost level we have embedded systems that implement the full communication architecture and are able to interact with all other systems of the architecture.

4.2. Properties

One major difference between COTS systems and embedded systems are the system properties. In most COTS systems the focus is on functional properties that directly correspond to the function that is expected from the system. Other properties, such as performance, are also of interest, but they are not the driving force during development.

For most embedded systems this is not true. The function and therefore the functional properties are also important, but they are only one aspect of the whole picture. Other properties which are not directly connected to the function of the systems have at least the same importance — we call these properties non-functional properties.

Examples are timeliness, performance, reliability, energy consumption and resource usage. These properties depend both on the application and the hardware of the device.

Regarding communication, non-functional properties are also a very important issue that has to be incorporated into each communication technique that interacts with embedded systems. Otherwise situations may occur where the device is not able to fulfill its original task. E.g., a battery powered device can only operate until the battery is empty. If wireless communication consumes twice as much energy than using the de-

1 Please note that we mean by "possibility to exchange information" also techniques such as a voltage on a cable.

2 This is true in general, but there are pacemakers that allow a doctor to control certain parameters.

3 By "proprietary" we here not only mean "non-standard" but also anything that cannot be used outside the specific application domain of the device, e.g., some embedded real-time communication standard such as CAN bus [1].

vice without it, then using this ability without considering the non-functional property "energy usage" could decrease the time of use to one third.

Therefore, our proposed contract based service architecture has to include these non-functional properties in a way that they have impact onto the communication process. This is only possible if their special needs are expressed in service contracts.

Now, we consider some non-functional properties as examples:

- *Timeliness* — Many embedded systems support hard real time which means they have to fulfill certain tasks within a hard deadline. Missing such deadline could result in a catastrophe or a loss of money. Therefore, communication with such systems is only possible in a way that does not violate the timeliness behavior of the system. If the communication itself is part of the real-time process, it also has to fulfill timeliness requirements (e.g., end-to-end delays) that depend on the application. All these properties have to be included into service contracts.
- *Fault tolerance* — Many embedded systems fulfill tasks that do not tolerate a down-time of the computer system. In such cases fault tolerance techniques such as redundancy in space or time are applied. Including such systems into service architectures requires to include this into contracts because communication must work in a way that it is compatible with the fault tolerance concept, e.g., messages have to be replicated to both subsystems.
- *Energy consumption* — Embedded systems are often part of battery powered devices where battery lifetime is critical to the application. Therefore, all parts of the application are designed for this goal. Regarding communication, this implies energy-aware communication schemes which can be expressed in service contracts.

5. Services for Embedded Systems

We propose to discover and access embedded systems using information stored in contracts. Depending on the type of the embedded systems that we want to expose as services, and on the location where the contracts are stored, we identify three types of services architectures: *proxy*, *translator* and *full* architecture.

Proxy architecture allows devices with proprietary physical communication to expose their functionality as a service. In order to achieve this, we must perform physical and logical conversion of the service request

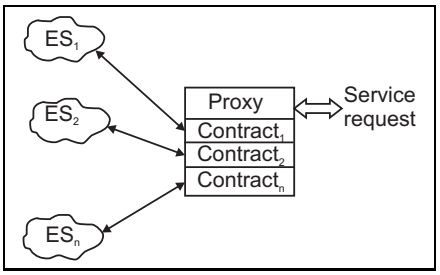

Figure 1. Proxy approach provides physical and logical conversion

(Figure 1), since embedded device communicates in a non-standard physical manner. We introduce a proxy, which has a knowledge of both service request message format and message format of the embedded system. The role of the proxy is to transform service request into the physical and logical format that an embedded system can understand. It receives a request from a standard physical medium defined by service architecture and converts it for a proprietary medium that an embedded system is using for communication. Furthermore, since it is obvious that proxy must be realized in hardware, we can cache embedded system's contract in the proxy's memory.

Translator architecture covers devices with proprietary logical communication. That means that although an embedded system uses standard communication, it does not understand service requests. Here, we introduce a translator (Figure 2), which is a middleware that intercepts service requests and translates them into a logical format that the embedded system can understand, without physical conversion, that is, it forwards the request on the same physical medium with different content. Since translator is very lightweight, it makes no sense to store contracts externally in translator. In this approach, contracts are stored by embedded devices themselves.

Proxy and translator architectures are intended for embedded systems with low resources, where it is not possible to realize additional service middleware on the system itself, or for systems where cost of adding those functionalities is to high. However, we allow for the third approach, which is a full service architecture, where an embedded system can directly respond to service requests (Figure 3). An embedded system employing full service architecture communicates in a standard prescribed way and can understand service requests. This architecture also allows for the intermediate directory that can store contracts, but purely for

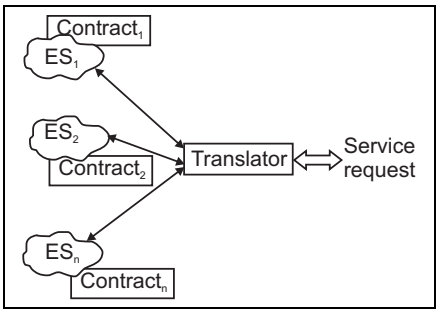

Figure 2. Translator approach provides logical conversion

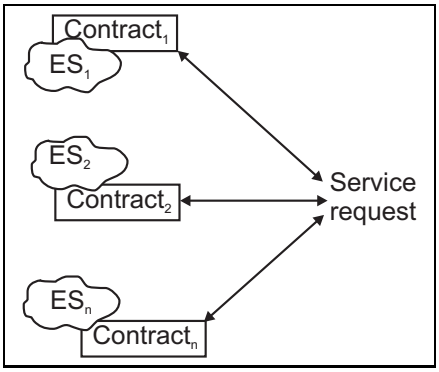

Figure 3. Full approach where embedded systems directly provide services in pervasive environment

the sake of facilitating service discovery, and not because embedded systems are not capable of managing service requests. Grouping contracts in a service directory is a decision that depends on the particular application. The architecture does not make directories mandatory, since in this case all communication can be realized peer-to-peer. Introducing directories, one can achieve better performance and organization, in a way that has already been solved for existing service architectures.

6. Conclusion

Some statistics report that over 98% of microprocessors manufactured today end up in embedded systems. These devices have certain computation power, memory and I/O capabilities. Still, they are primarily used for task-specific products that do not utilize their full power. We view all embedded systems that we employ today, and that we will use tomorrow, as a pool of resources. If we could propose a schema that would enable custom and easy access to various functionalities provided by different kinds of embedded systems that are all around us, we would be able to combine the power of embedded systems and come up with a whole range of applications that do not exist today.

The concept that we explore is similar to one of grid computing. We want to link all available embedded systems in a network of services, that are available on demand. In order to provide such a complex architecture we propose a service architecture based on contracts. Contracts are descriptions of systems that encompass functional and non-functional properties. Using contracts, embedded systems can advertise their functionalities, and clients (that can be other embedded systems) can discover and use them. Since embedded systems cover a vast number of devices of different capabilities, we propose proxy, translator and full service architecture that, in our opinion, provide optimal resource usage depending on device capabilities.

Using proposed approach, a whole range of possible applications emerge: from personal networks of wearable devices, smart houses, and automotive applications, to concepts such as pervasive and ubiquitous computing, where embedded systems discreetely fade into the background and users just consume the services they offer.

References

[1] *CAN Specification Version 2.0.* Robert Bosch GmbH, Stuttgart, 1991.

[2] OSGi Alliance. Osgi service platform specification overview. *http://www.osgi.org/resources/spec_overview.asp*, 2003.

[3] Ian Foster, Carl Kesselman, Jeffrey M. Nick, and Steven Tuecke. The physiology of the grid. *http://www.gridforum.org/ogsi-wg/drafts/ogsa_draft2.9_2002-06-22.pdf*, 2003.

[4] Hermann Kopetz and Neeraj Suri. On the limits of the precise specification of component interfaces. In *Proceedings of The 9-th IEEE International Workshop on Object-oriented Real-time Dependable Systems (WORDS 2003F)*, Italy, 2003.

[5] Bertrand Meyer. Contracts for components. *Software Development*, 2000.

[6] Sun Microsystems. Jini network technology. *http://wwws.sun.com/software/jini/*, 2003.

[7] Nikola Milanovic, Vladimir Stantchev, Jan Richling, and Miroslaw Malek. Towards adaptive and composable services. In *Proceedings of the IPSI2003*, Sveti Stefan, Montenegro, 2003.

[8] W3C. Web services architecture. *http://www.w3.org/TR/2003/WD-ws-arch-20030808/*, 2003.