2010 IEEE 24th International Conference on Advanced Information Networking and Applications Workshops

Service Migration Scenarios for Embedded Networks

Stephan Sommer, Andreas Scholz, Irina Gaponova, Alois Knoll, Alfons Kemper
Technische Universität München
Boltzmannstr. 3, D-85748 Garching, Germany
{sommerst,scholza,knoll,kemper}@in.tum.de

Christian Buckl, Gerd Kainz
fortiss GmbH
Guerickestr. 25, D-80805 München, Germany
{buckl, kainz}@fortiss.org

Jörg Heuer, Anton Schmitt
Corporate Technology, Multimedia and Network Communication, Siemens AG
D-81730 München, Germany
{joerg.heuer,anton.schmitt}@siemens.com

Abstract

More and more devices are becoming network enabled and are integrated within one large, distributed system. The service-oriented paradigm is the predominant concept for the implementation of this approach and helps to deal with heterogeneous network infrastructures. For long term deployments, e.g., in process and building automation, installations have to be adaptable over time. During runtime new services will be added to the network, old services will be replaced and deployed services will be relocated to adapt the network to environmental changes and new application fields. Handling these updates is challenging, because the impact on the currently executed applications has to be minimized. In general, such tasks are very application specific. However, the service based application development paradigm allows applying a generic approach for most of the scenarios. This paper presents such a generic solution by adding mechanisms, which are capable of handling service migration and service updates, to a sensor network middleware.

1 Introduction

Using sensor networks in industry has a long tradition. Where in the past the sensor networks were dedicated for one specific measuring task and each vendor deployed his own proprietary network as a bundled solution comprising hard- and software, the trend for future deployments shows that multifunctional networks are needed to reduce management and infrastructure cost. Using one communication infrastructure also allows reducing the amount of sensors deployed: sensors can be re-used for further applications without additional overhead for wiring, independent of the vendor and the application it was firstly deployed with. This saves costs for hardware acquisition and for infrastructure management. Connecting sensors to a single network (probably also consisting of different subnets using different communication technologies) and making them accessible by different applications can be seen as a trend for new deployments. Because most of these networks are also connected to the Internet, devices are becoming accessible with Internet technologies, leading to the *Internet of Things*.

Service-oriented architectures (SOA) are used to lower development cost and to support modularization of applications. Web services are a well known implementation of a SOA and can be found in the Internet domain. Due to resource constraints, the Web service technologies cannot be directly mapped to the domain of embedded system. The ultimate goal is to support SOA concepts in the embedded domain by using a suitable implementation. Different approaches for reaching this goal have been suggested in the literature. The main concept is to restrict the functionality of the middleware, e.g. by removing run-time support to dynamically detect services within the network, or to use long term service compositions, instead of ad-hoc service interaction to avoid the overhead of frequently initializing the service interaction. These modifications reflect the behavior of embedded systems which are rather statically config-

978-0-7695-4019-1/10 $26.00 © 2010 IEEE
DOI 10.1109/WAINA.2010.84

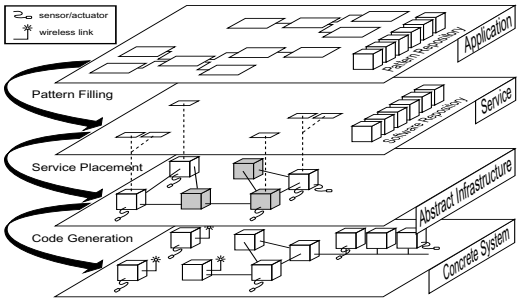

Figure 1. Embedded Network Views

ured and where external tools are applied to reconfigure the network. Examples for such approaches are the ϵSOA[10] project or the Devices Profile for Web Services (DPWS)[6]) specification.

Considering industrial applications for process monitoring or building automation shows that these applications consist of many different services and need to be used for many years or even for decades. For such long term deployments, flexibility is a major concern. It is necessary to support updates of the deployed services as well as the migration of already deployed services to new hardware without a complete system interruption. Updating or migrating services is usually a very application specific task, but in sensor networks, where applications consist of interconnected services, a generic approach is feasible for most scenarios. This paper will present a migration workflow suitable for embedded systems to solve this issue. The main contribution of this work will be the classification of different migration scenarios and the identification of the main components required to implement the migration.

The approach is currently restricted to systems with **real world awareness** and **soft real-time** applications and is not directly applicable for **hard real-time** systems. For our migration workflow we assume, that most of the reconfigurations are fast enough to be performed between two messages. Otherwise, the middleware needs to guarantee that messages can be buffered, that messages are not delivered twice and that critical reconfigurations (e.g. it they involve multiple services) are performed as a transaction.

The paper is structured as follows: in the beginning, we give a short summary of the ϵSOA platform for better understanding of the infrastructure we are targeting. In section 3, we elaborate the different scenarios for service migration, as well as the challenges for this task. Our implementation of a migration workflow and a suitable demonstrator is shown in section 4. Finally we summarize related work and conclude the paper with a discussion about topics for ongoing and future work.

2 ϵSOA

Before we show our migration workflow, we first want to introduce the ϵSOA platform, which is our development platform for embedded networks. The main motivation was to combine a well known development approach - model driven development - with concepts taken from Service oriented Architecture and actor-oriented design to ease the development process of heterogeneous, highly-distributed embedded systems.

2.1 Actor Oriented Design and SOA

As known from the web service domain, a service oriented architecture provides many advantages for application development and maintainability like well structured applications consisting of interacting services, each of them implementing only a single aspect of the application. When developing software for the embedded domain, one also has to deal with hardware interactions, such as reading from a sensor or writing data to an actuator. To formalize such a control application, it is a common approach to use an actor oriented design[8]. Actors can represent sensors (inputs), actuators (outputs) and control functions; the interaction between actors is realized using ports. For our approach, we use actors based on the concepts common in the domain of embedded systems as services for the service-oriented architecture. Actors/services can be sensor-, actuator- or control-services which communicate by sending data from outputs of one service to the inputs of another service.

In contrast to Web based SOAs which are often based on an ad-hoc request-response message pattern, control applications are typically event/data driven: the data is acquired by sensors and published to all connected services. These connected services can be actuator services triggering a hardware action or control services which can produce new output data based on their inputs. Therefore, instead of using ad-hoc interactions, the interaction is defined statically in ϵSOA and is performed in the fire and forget scheme. By using our development tools [10], end-users can modify the interaction between services by adding or deleting edges between the ports of services at run-time. However, we assume that these modifications occur seldom in comparison to the communication between services.

2.2 ϵSOA Middleware Design Principles

Our ϵSOA middleware is based on a hierarchy of three views of the embedded network, as depicted in Figure 1 to separate concerns and to make the development process more clear. The Application view contains an overview over all installed applications. Applications are built based on patterns that specify the required services on an abstract

level. This abstraction allows using soft- and hardware from any vendor, as long as these components provide a compatible interface. On the more fine grained Service view, the applications consist of various interconnected services which form the logical structure of the distributed application. Based on information provided by the Abstract Infrastructure view, the execution of applications is optimized by deciding where a specific service should be installed and by adapting the middleware running on the nodes. These optimizations are based on the characteristics of the underlying hardware, which are extracted from the Concrete System and annotated to the system model in the Abstract Infrastructure view. A detailed discussion of these different layers can be found in [10].

An important design criterion during the development of this layering was to provide suitable views for all parties involved in the development process of an embedded network. An application developer will only have to focus on connecting the right services to provide and process information whereas a service developer will only have to focus on the control logic in his service. The communication, service management and monitoring of the whole network is done by our ϵSOA middleware which can be tailored using a model-driven development tool[2].

3 Migration Scenarios

Sensor networks tend to be used for long period of time in large industry deployments. Additional devices or even additional sub-networks often need to be deployed to adapt the network to new challenges emerging over time. To face this problem, it is a common way to use the communication standard already deployed and simply add new dedicated hardware nodes with the required functionality.
In our approach focusing on a middleware for heterogeneous service oriented architectures, a further way of extending the application is possible. As described in more detail in section 2 our applications consist of services distributed over the network. If such a network should support additional applications, new services can be easily installed on existing nodes to form new applications in cooperation with the already deployed services and applications. Adding these new services to already existing applications or replacing services in an already deployed application is a critical task because the interactions of different applications have to be taken into account.
A further critical task in these networks is to move already deployed services to different nodes. Possible reasons for this service shifting are better load balancing for the network traffic, equal resource utilization on the nodes and links, and finally avoidance of energy depletion of battery powered nodes. In the following, we will elaborate the different scenarios for such reconfigurations and point out the consequences for the update process.

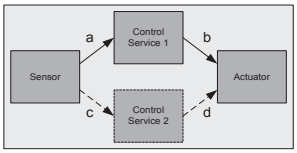

Figure 2. Simple Application

The term $state$ comprises in our definition all (configuration and runtime) information locally stored in a service necessary to process incoming data. For our applications, the state can only be changed by data received from ports or read from the hardware.

3.1 Extension of already deployed applications

Extending an already deployed application by a new, additional service is the easiest update scenario. In this scenario, the new service requires data already provided by one service of the old application. It is only necessary to deploy the new service to the network and to connect its input ports to some of the output ports of the already deployed application. Before deploying the service, possible changes in network utilization and resource consumption need to be considered. For this, the same workflow can be used as for the initial service placement, but with fixed placements for the already deployed ones. The deployment can be done at runtime without harming the already deployed application if the new service can be instantiated at runtime (depending on the platform), or if the new service is deployed to a node not involved in any application until now.

3.2 Service Migration without explicit state transfer

A second scenario for service migration is the replacement of an already deployed service by a new one. First we only consider a migration where no inner state of the service needs to be transferred, either because the service is stateless or because the new service can automatically recover the internal state. This could be the case for very simple services like data converters or basic logic operators and for those services, which can acquire the state over time just by listening to input data[1] like services calculating the average of the last x values. Afterwards, we will extend this scenario for state full services.
The first step for the migration or replacement [2] of a service is to deploy the new service to an adequate node. As already

[1] This interface has to be implemented by the service developer.

[2] A migration can also be seen as a replacement of a service by a new one of the same kind.

mentioned for the first scenario, the placement can be done using the already available tools. A very simple application is depicted in figure 2. This applications consists of a source service (Sensor), a control service (Control Service 1) and a destination service (Actuator). The $ControlService2$ is the service which replaces the $ControlService1$.

3.2.1 Stateless services

For stateless services, the migration is almost completed at this point. The only remaining task is to remove the data paths connected to the old service and add connections for the new one. Usually, connections from a source service which provides the data to the service, and further connections from the output ports of the service to all data subscribers exist.
The best way to perform this task is to add the new connections for the newly deployed service beginning from the sink side to the service (figure 2, connection c) and, after that, from the new service to his data recipients (figure 2, connection d). The removal of the connections involving the $ControlService1$ is done vice versa (figure 2, connection b and a).

3.2.2 Stateful services

For state full services that do not provide an interface for state migration and for services which can acquire the current state only by listening to the data flow, additional tasks need to be performed before the migration is complete. The first steps are the same as for stateless services until the connections from the new service to the subscribers are added. Before these connections can be configured (figure 2, connection d), the process has to be a halted until the correct internal state of the new service ($ControlService2$) is acquired. After this service is *up to date*, the remaining data paths can be added from this service to all the subscribers. At the subscribing nodes, the reconfiguration (removal of the old data paths related to the old service (connection b) and the addition of the new data paths for the new service (connection d)) needs to performed in a transactional way. If a message arrives in this very short transaction phase, it has to be cached to avoid possible application misbehavior.

3.3 Service Migration with state transfer

The approach already described for stateless services, can be extended to handle the migration of state full services. Handling the state transfer also comes with coordination challenges for the reconfiguration of data paths. As shown for the case without explicit state transfer, the data paths need to be adapted in a coordinated way to integrate the newly deployed service.
To guarantee a seamless and consistent migration, it is important to add the data paths from all sources to the new service in exactly the moment where the state transfer begins. After that point in time, the old service does not receive further data until the state transfer is completed. For the new service receiving the state, it is important to not start processing of inputs before the state transmission was completed. If the application is e.g. a climate control system in building automation and some temperatures measured are lost (assuming a high enough data acquisition rate) it will no harm the application. But if we consider data loss in an application with very rare events or a user interaction e.g. a user pushing a button, the application and the real world could get inconsistent without buffering the messages.
If the service to migrate is involved in many different applications, all requirements of the involved applications need to be taken into account. For example some of the involved services might tolerate the data loss while others do not, some can tolerate downtime while others cannot. In the worst case all applications related to a service which needs to be migrated need to be stopped.
According to the challenges elaborated for the service migration in this section we developed a workflow, which is presented in the next section.

4 Migration Workflow

Migrating services inside an embedded network is implemented in our project in a stepwise approach. First of all, an instance of the service needs to be created at the destination node. The way this instance is created depends on the runtime and the underlying system infrastructure.
After a new instance is created at the destination host, the internal state needs to be transferred to the new instance. The state transfer is split up into four phases namely *serialization*, *state transfer*, *de-serialization* and *reconfiguration*. In the process of state transfer two components are involved in addition to the source and destination service. These components, namely the $MigrationCoordinator$ which initiates the migration, and the $MigrationFacilities$, which actually performs the local operations, necessary for the migration at the nodes. An example scenario is depicted in figure 3. In this scenario, the service instance x is moved from the node B to node C (x'). All management extensions for the middleware namely the application repository, the facilities for network management, and the $MigrationCoordinator$ are located at node A in this example. The interaction of services is stored in the application repository; information related to the network is stored in the network management, which subscribes statistics from the nodes (using the middleware) in the network and processes them to provide e.g. utilization statistics. Al-

though information is gathered by the system, a migration is only triggered by the user / administrator.

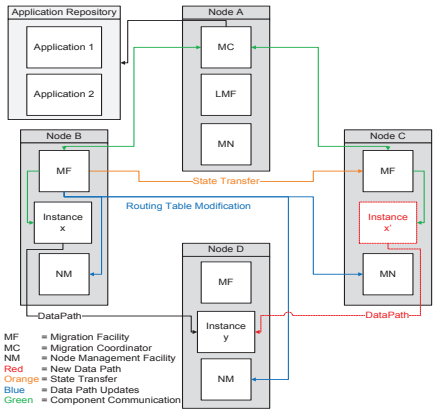

Figure 3. Migration Scenario

At the beginning of the process, the source service serializes its inner state and provides it to the migration facility located on the source node. The migration facility then transfers the state data to the corresponding migration facility on the destination node. The migration facility finally provides the data to the destination service using the migration interface where the data is de-serialized.

After transferring the internal state and starting the new service, connections using the old service need to be replaced by connections using the new service. The last step finally is to decommission the old service.
Both components, the Migration Director and the Migration Facilities are generic and interact with the services using predefined interfaces. In the following, the components will be explained in more detail

4.1 Migration Coordinator

In our middleware, the software parts responsible for the migration are split into two different kinds; the first one is the *Migration Coordinator*, which is the centralized part, and the *Migration Facilities*, which need to be installed at least on the nodes involved in the migration process.
The Migration Coordinator is responsible to coordinate the migration according to network and application needs. To fulfill this task, the migration coordinator has in-depth knowledge of the applications, services, requirements and the data paths of the network. It gathers this information from the network management facility installed on one or, if a distributed implementation is used, on several nodes in the network.
When a migration is triggered by the user or by a monitoring agent, the coordinator first checks if the source and destination nodes are available in the network and if the requested service is installed on the source node. Using the meta data dictionary located in the network management facility, the information is gathered if the destination node can handle the service and if the state migration is supported by the service or not. In case of problems, the user gets notified and the process is stopped here.
If all pre-conditions for the migration are met, the migration coordinator triggers the instantiation of the destination service instance at the destination node. The instantiation itself is done by a middleware component and can, in worst case, require to overwrite the complete software image on the destination. This can of course effect the remaining applications being executed on the node or transmitting data using this node as a hub.

4.2 Migration Facility

In contrast to the migration coordinator which is a centralized component, a migration facility is located at each networked node which provides support for service migration. The migration facility can perform two different tasks according to the responsibility in the migration process. The migration facility on the source node is responsible for checking if the requested source service is available and if it implements the required migration interface. If the interface is implemented, the migration facility requests the service state. This state information is then stored and, according to the information provided by the migration coordinator, sent to the migration facility at the destination host.
The migration facility at the destination node receives the system state. It checks if the destination service is instantiated properly and transfers the state using the migration interface. Finally it starts the new service and notifies the migration facility. For monitoring purposes, this message is also forwarded to the migration coordinator.
To finish the migration process, the data paths reconfiguration is triggered by the migration facility at the source node. If the reconfiguration was performed properly, the application can be resumed.

4.3 Implementation

The practicability of our approach elaborated in this paper was implemented and tested in our ϵSOA demonstrator. The demonstrator is described in [3]. Basically it illustrates an energy scenario which consists of a fridge, some lamps and a rechargeable battery. To control the devices there are several ZigBee nodes executing the TinyOS version of our middleware, some more powerful nodes executing the JAVA version of our middleware and a PC node connected via a web service bridge for visualization. All these components form the embedded network.

To increase the flexibility of this scenario, it was necessary to support reconfiguration of applications, where the relocation of services is a key part. We could show for different test cases that stateful services can be successfully moved between several nodes

5 Related Work

Within different application domains, standardized middleware architectures, e.g., KNX[7] for the building automation domain or AUTOSAR[1] for automotive applications were established. Because of their low level of abstraction they can only provide a communication layer for a specific application domain, but do not really focus on how to interconnect applications of different domains. A further component based middleware for distributed applications is CORBA[9] which provides great functionality for state and component migration, but was designed to interconnect heavy weight application servers, where our approach targets resource constraint embedded systems. Nevertheless, the concepts implemented in CORBA can be seen as an inspiration, even for small devices. Other projects which apply the service oriented approach are MORE[11] and RUNES[5]. Although they also provide functionality for service migration, we believe that our approach with application patterns and a rich set of meta information allows a more user friendly network management.
Moving currently executed code between different systems is a challenge also encountered in the domain of system virtualization. A typical approach in this scenario is to copy the whole system image (including the RAM) to a new host and update the network configuration after completion[4]. This approach cannot be used in systems using heterogeneous network technologies, which are common in embedded networks (e.g. a mixture of wired and wireless links network using different network protocols). Additionally, the resource constraints in embedded networks prohibit some of the common solutions, such as storing the whole RAM image at the original host and forwarding it to the new host upon completion of the migration.

6 Conclusion and Future Work

In this paper, we elaborated the challenges for service migration in embedded networks. We proposed a workflow for the migration of services using a decentralized component, the *Migration Facility* and a centralized component, the *Migration Coordinator*. Based on these generic components, a centrally controlled and monitored migration of distributed services with the scalability of a distributed approach can be achieved. The main contribution of this work was the classification of different migration scenarios and the identification of the main components required to implement the migration.
In the future, we will investigate on service dependencies and how such dependency graphs could be partitioned according to the migration constraints defined by the service and application developer. A further topic will be to demonstrate the feasibility of our approach for single image based TinyOS nodes with a pre installed service library and for scenarios where the whole runtime needs to be updated. This will also include performance measurements to identify potential bottlenecks and according solutions.

References

[1] AUTOSAR – Automotive Open System Architecture. http://www.autosar.org/.

[2] C. Buckl, S. Sommer, A. Scholz, A. Knoll, and A. Kemper. Generating a tailored middleware for wireless sensor network applications. *Sensor Networks, Ubiquitous, and Trustworthy Computing, International Conference on*, 0:162–169, 2008.

[3] C. Buckl, S. Sommer, A. Scholz, A. Knoll, A. Kemper, J. Heuer, and A. Schmitt". Services to the field: An approach for resource constrained sensor/actor networks. In *The Fourth Workshop on Service Oriented Architectures in Converging Networked Environments (SOCNE 2009) – extended version*. IEEE, May 2009.

[4] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield. Live migration of virtual machines. *Networked Systems Design and Implementation*, 2005.

[5] P. Costa, G. Coulson, R. Gold, M. Lad, C. Mascolo, L. Mottola, G. P. Picco, T. Sivaharan, N. Weerasinghe, and S. Zachariadis. The runes middleware for networked embedded systems and its application in a disaster management scenario. *Pervasive Computing and Communications, IEEE International Conference on*, 0:69–78, 2007.

[6] Devices Profile for Web Services. http://specs.xmlsoap.org/ws/2006/02/devprof/devicesprofile.pdf.

[7] KNX the Worldwide STANDARD for Home and Building Control. http://www.knx.org/.

[8] J. Liu, J. Eker, J. W. Janneck, X. Liu, and E. A. Lee. Actor-Oriented Control System Design: A Responsible Framework Perspective. *IEEE Transactions On Control Systems Technology*, 12, No. 2, March 2004.

[9] Object Management Group. Common object request broker architecture (corba) specification, version 3.1, Jan 2008.

[10] A. Scholz, C. Buckl, S. Sommer, A. Kemper, A. K. an d Jörg Heuer, and A. Schmitt. eSOA - service oriented architectures adapted for embedded networks. In *Proceedings of the 7th International Conference on Industrial Informatics)*, June 2009.

[11] A. Wolff, S. Michaelis, J. Schmutzler, and C. Wietfeld. Network-centric middleware for service oriented architectures across heterogeneous embedded systems. In *11th International IEEE EDOC Conference, Middleware for Web Services Workshop*, 2007.