A Distributed Scaleable Real-Time Add-On for Operating Systems

Uwe Brinkschulte Holger Vogelsang

Institute for Microcomputers and Automation
University of Karlsruhe
Haid-und-Neu-Str. 7, 76131 Karlsruhe, Germany

Abstract

This paper describes the architecture of a distributed, scaleable real-time system platform. After defining the requirements, the basic conceptions for the service oriented system platform called OSA+ are discussed. The main concept is the exchange of tasks and results. This mechanism is used for communication, synchronisation and definition of real-time constraints between services.

1 Introduction

The main research area of the IMA is the design and development of complex automation systems. One of our interests focuses on automated guided vehicle systems (AGVS) for industrial applications. Because of individual work- and transportation-flows, an AGVS for a specific plant is a unique system. It differs in hard- and software components from a system designed for another plant.

To allow an efficient and cost-effective development, test and maintenance of such individual systems, we have decided to use an open service-oriented architecture based on a scaleable system platform.

The following sections describe the conception, requirements and the resulting architecture of such a system platform for individual distributed automation systems.

2 Conception and Requirements

Figure 1 shows the conception of an open service-oriented architecture. This architecture consists of services, applications and an interconnecting system platform. The system platform has two main tasks:

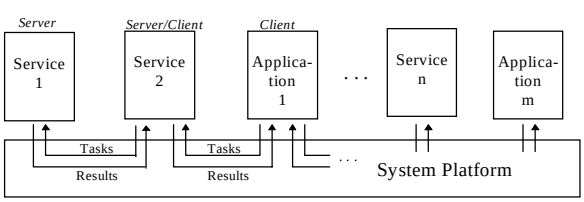

Figure 1: Service-oriented architecture

- ***Service definition***
  provides the capability to plug in services and applications into the system platform and to configure and connect those services and applications.
- ***Service access***
  allows the access to the plugged-in services. It manages the communication between services and applications.

In this conception, service access is done by processing tasks and results. Tasks and results are used for communication and synchronization between services and applications. Even multirole components (server/client) are possible.

With a well defined, documented and user-friendly interface for these two tasks, the system platform allows a flexible construction of individual open systems. Basic services (e.g. database service, man-machine service, measurement and control service, ...) can be combined with completely user-definable services and applications.

Furthermore, the system platform abstracts from the underlying hardware, operating system and the used programming language. It provides platform independence.

Finally, the system platform hides distributed hardware. Local system platforms combine to a global virtual system platform. For applications and services it

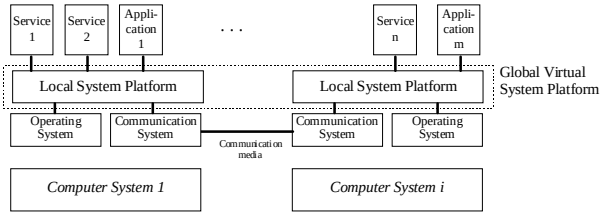

Figure 2: Distributed hardware hidden by the virtual system platform

is completely invisible, if they reside on the same machine or on distributed machines. It should be even possible to change distribution at runtime without changing any service or application.

To use the system platform for AGVS or similar automation systems, it must meet some special requirements:

- ***Scaleability***
  The system platform should be scaleable for:
  - **Different hardware**
    The system platform must operate with the same interface and behavior on very different hardware platforms from simple microcontroller systems to workstations and PC's.
  - **Different operating systems**
    The system platform must be adaptable to different underlying operating systems (or no operating system at all).
  - **Different communication systems**
    In distributed applications, the system platform must be scaleable for different underlying communication systems (serial interfaces, field busses, networks, ...)
  - **Different process models**
    It should be possible to use different process models for the services plugged in the system platform. A service may e.g. operate as lightweight processes (thread), heavyweight process or as a sequential procedure.

- ***Real-time behavior***
  In automation applications, real-time behavior must be guarantied for time-critical services. So, a service must be definable as real-time or non real-time service. For real-time services, guarantied and predictable timing must be provided for:
  - **Service access**
    Transportation times for tasks and results must be defined or better, be definable by the user (e.g. earliest and latest time for a task to reach a service).
  - **Service scheduling**
    Execution times for real-time services must be predictable and controllable (e.g. using fixed priorities , deadlines, critically-values, ...)

  Of course, the real-time behavior of the system platform depends on the underlying hardware, operating system and communication system. For example, on a system platform with underlying real-time operating system but non-real-time communication system, only local services can be real-time.

- ***Remote configuration, test and maintenance***
  Figure 3 shows an example of remote test and maintenance.

In industrial automation systems, the ability of remote system configuration, test and maintenance via long distant lines saves costs and time. The system platform can significantly simplify this, if it provides the following features:

- **Service Movement**
  A service can be moved at runtime from one machine to another machine.
- **Service Replacement**
  A service can be replaced at runtime by another service

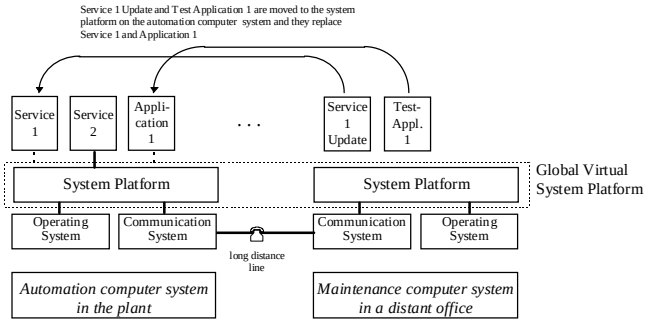

Figure 3: Remote test and maintenance on a virtual system platform.

3 The OSA+ system platform

The development of the system platform OSA+ is guided by the requirements analysis in chapter 2. OSA+ picks the idea of services shown in figure 1. We will now discuss the following points, using the platform functions as a principle.

- Operation modes and functions
- Scaleability
- Real-time capabilities, scheduling

3.1 Platform Initialization

A system platform is initialized with several parameters, which influence the later behavior of the system.

*osaInitPlatform(char *platform, osaPartner *partner,*
osaNetworkType network,
*osaProcInfo *info);*

The parameters control the kind of communications system(s) to be used, the names of the partner platforms, which form the global virtual platform and the name of the own platform. The last parameter returns the capabilities of the system platform together with its state. This represents the first step of scaleability.

3.2 Service Creation

The purpose of this function is the creation of a new service with several properties:

*osaStartService(void(*Proc)(), osaProcType type,*
*osaConfigType *config, char *name);*

The parameters *type* and *config* represent the second step of scaleability. The parameter *type* defines the process type used. Possible values are procedures (as a process substitute), lightweight processes and real operating system processes. Whereas the first two types force shared memory between these kinds of processes on the same platform, the latter uses operation system processes with separate memory segments. The value of *config* sets additional configuration parameters for the service, e.g. real-time behavior. During service creation, OSA+ distinguishes only between real-time services and non-real-time services. Specific real-time parameters are bound to service operation described in the next section.

Of course, the grade of available real-time capabilities and process types depend on the underlying operating- and communication system. This is one of the information returned by the *info* parameter of *osaInitPlatform*.

3.3 Service Operation

This section contains the modes and principles of service operation. The idea behind this is much different from any other concepts found. It allows a very flexible service configuration and time specification with only a few functions. Services interact by the exchange of tasks and results. There are no other communication or synchronization methods necessary. As an introduction, Figure 4 shows a simple example: Service A and B communicate synchronized using tasks and results.

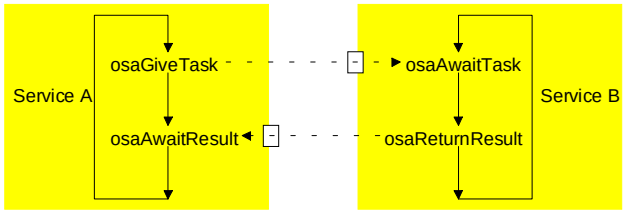

Figure 4: Communication example

*osaGiveTask(osaTask *task);*
*osaReturnResult(osaResult *result);*

The main idea is to use this exchange of tasks and result also as the only way to add time conditions to services.

Therefor, tasks and result can contain optional deliver modes to specify time restrictions. If no deliver mode is given, all tasks and results are handled with the same priority, using the first-come first-serve method. The deliver mode itself is a structure with the following members:

```
typedef struct {
        struct {
                osaTime         deliver_at,
                                deliver_tolerance,
                                deliver_ period;
                osaLong         deliver_mult;
                } order_mode;
        union {
                struct {
                        osaTime         deadline;
                        osaShort        critically;
                        } realtime;
                struct {
                        osaShort        priority;
                        osaTime         timeslice;
                        } realtime_priority;
                struct {
                        osaShort        cpu_load;
                        }realtime_load;
                } service_mode;
        } osaOMode;
```

The following description uses the expression *message* to specify either *task* or *result*. We distinguish between messages, influencing the deliver mode of themselves (*order_mode*) and messages, influencing the behavior of the destination service (*service_mode*). Both modes can be combined for the most flexible delivery.

1. ***order_mode:***
   If the structure *order_mode* is specified, this controls the real-time behavior of the service access, which is the mode, the message is delivered with. The attributes of this mode are defined as follows:
   - *deliver_at* and *deliver_tolerance* give the interval, in which the message must be delivered. If this time is exceeded, the message is returned to its sender together with an error message.
   - The system platform is able to handle standing orders. *deliver_period* contains the time interval, after which a message must be send again. *deliver_period* is ignored, if *deliver_mult* is *1*.
   - *deliver_mult* acts as a counter and specifies the number of times, the same message has to be delivered.

2. ***service_mode:***
   This controls the real-time behavior of the service scheduling (see requirements analysis). Three different kinds of modes allow a very flexible configuration of the behavior of the destination

service. This allows the sender of a message to control the behavior of the receiver with respect to the importance of the message.

realtime:

- *deadline* contains the last possible point of time, at which the answer to the message must be delivered.
- *critically* tells what to do, if deadline expires (soft-, firm-, hard-deadline). *deadline* and *critically* are used as well to calculate the new priority of service, which receives the message.

realtime_priority:

- *priority* is an alternative way to specify a simpler form of time constraint. *priority* contains the absolute fixed priority for the receiving service, used by the scheduling algorithm.
- The timeslice, a service can use before it is preempted, is specified using *timeslice*.

realtime_load:

- *cpu_load* contains a percentage of CPU usage for the receiving service. This allows the service-execution nearly independent of the current CPU load.

In most systems, services should be able to set their own priority and time behavior. This can be realized very easy using task and results: To be started with a given interval and priority, a service sends itself a standing order or changes previous orders to itself. With this very simple mechanism, services can control themselves and other services only by sending orders or result. This allows maximum flexibility with a minimum number of functions. The sender is able to change the real-time conditions of the receiver for one or more messages. Figure 5 shows all variants to control the time behavior of services and message deliveries.

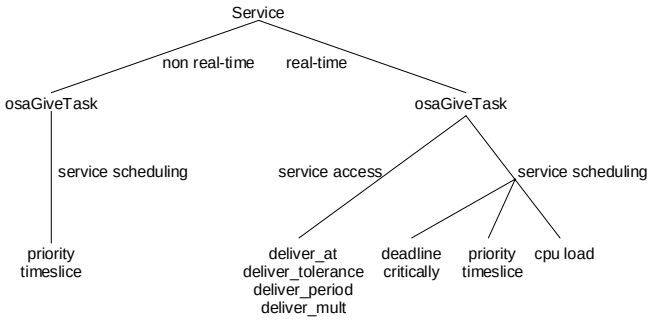

Figure 5: Time control

After the discussion of time constraints, we will have a look at the different kinds of message deliveries. The first and simplest way to define the destination or partner of a message is to write its service number into the task or result. The system platform can easily transport this message to this destination. Sometimes it is not necessary to specify a unique partner, sometimes it is sufficient to specify a class of partners. In this case, the system platform is responsible of choosing a matching partner. There could be some strategies: The simplest is to take partners, connected over the fastest communication systems, partners, who are idle, partners on fast systems, ... In our approach we choose the nearest idle partner. The system platform transports the message to this partner, if only the destination class is given in the message. As a consequence, messages to the same class of services are not necessarily handled by the same service. And how are classes build? The construction of classes is simple: All services with the same name are in the same class. They can be distinguished only by their unique process number.

There are - of course - more functions to control services, to handle the communication, to bundle services to libraries and to exchange services at runtime for remote configuration, but we can not discuss them here. A more detailed description can be found in the internal paper of the system platform [SP97].

4 Conclusions

The development of the OSA+ system platform is an ongoing process. We have a working prototype with a subset of the total functionality. This prototype is in test with several applications and produces encouraging results. We hope to get the full functionality during this year.

References

[1] Uwe Brinkschulte, Holger Vogelsang, *The OSA+ System Architecture*, Internal Paper, 1997, http://www-ima.ira.uka.de/mitarbeiter/vogelsang/