Real-Time Service-Oriented Architectures: A Data-Centric Implementation for Distributed and Heterogeneous Robotic System

Pekka Alho, Jouni Mattila

Real-Time Service-Oriented Architectures: a Data-Centric Implementation for Distributed & Heterogeneous Robotic System

Pekka Alho[1], Jouni Mattila[1]

[1] Tampere University of Technology, Dept. of Intelligent Hydraulics and Automation, Finland
{pekka.alho,jouni.mattila}@tut.fi

Abstract. Cyber-physical systems like networked robots have benefited from improvements in hardware processing power, and can facilitate modern component and service-based architectures that promote software reuse and bring higher-level functionality, improved integration capabilities, scalability and ease of development to the devices. However, these systems also have very specific requirements such as reliability, safety, and strict timeliness requirements set by the physical world, that must be addressed in the architecture.

This paper proposes a real-time capable service-oriented architecture, based on data-centric middleware and an open real-time operating system. A prototype implementation for a robotic remote handling scenario is used to test the approach. The architecture is evaluated on the basis of how well it fulfils the expectations given for the service-orientation, including: reusability, evolvability, interoperability and real-time performance. In one sentence, the goal is to evaluate the benefits of a data-centric approach to service-orientation in a performance-critical and distributed system.

Keywords: real-time, distributed, SOA, data-centric, middleware, robotics

1 Introduction

Developing software for cyber-physical embedded systems such as networked robots is a demanding task, due to complex functionality that has to be realised in a distributed and heterogeneous computing environment which typically has requirements for real-time performance and fault tolerance. Many of the challenges in these systems are related to interoperability and growing scale. Typically a distributed control system will consist of several subsystems running on different platforms that produce and consume increasing amounts of data.

On a higher abstraction level, business processes are also becoming strongly networked to improve efficiency by automatically transferring data, task requests etc. between systems. This means that robotic systems must be integrated to operations management systems, open for external connections, and able to connect and cooperate with other machines. These requirements and challenges are not unique to robotics

– other domains like industrial automation, mobile machines and telecommunication have very similar issues.

Service-oriented software engineering has evolved from component frameworks and object orientation to meet the demands of more open and networked environments. It promotes reuse by decomposing business processes into reusable core services. The main benefits of service-oriented architecture (SOA) include high level of decoupling – provided by the service model – and interoperability which enables service providers and consumers to exist on different platforms. Two major downsides typically associated with SOA are complexity of developing such a system and increased overhead caused by communication mechanisms [6]. The latter is also related to the lack of performance guarantees, and presents a major challenge especially for embedded systems. An SOA implementation for robotic system therefore needs to place heavy emphasis on solving this problem, which is one of the key design goals for the architecture presented in this paper.

Application of SOA design principles to real-time systems (RTSOA) is a research topic that has come up in the last decade, with research including experimental implementations [3], [9], [10] and related key features like service composition [2], [4]. However, most of the current RTSOA approaches are based on the existing message-based Web Service standards. Web services face challenges when used in embedded systems, as messages need to be serialized in real-time [2], and quality of service (QoS) must be managed at the transport layer. Other challenges include complexity of networking with HTTP, XML, and SOAP; constraints imposed by embedded system architecture; and verbosity of HTTP and XML.

We believe that the service-oriented approach may be beneficial for the development of cyber-physical systems, but there is a need to test out different implementation solutions that fulfil the specific limitations and requirements of the target domain, including reliable communications, limited resources, and deterministic behaviour. In this paper we present a data-centric approach to RTSOA and evaluate it by implementing the proposed reference architecture for a robotic remote handling scenario. Remote handling involves human operators remotely controlling robots that perform tasks like maintenance or construction in dangerous environments, so reliability and performance of the system are vital for successful task completion.

2 Real-Time Service Orientation for Robotic Systems

2.1 Design Goals

We see the following as the main design goals for the real-time service-oriented architecture:

- Promote software **reuse** by producing reusable and decoupled software modules.
- Enable **composing** a working system out of reusable and existing services.
- Improve **interoperability** of heterogeneous systems (platform & programming language independence).

- Ensure **evolvability** [8] in the future; the architecture should support changing requirements and operating environments during the system lifecycle.
- Deterministic **real-time performance,** despite dynamically changing environment.
- **Dependable and fault tolerant** operation.
- Improve **cost-efficiency & ease of development;** the implementation should be able to use off-the-shelf solutions for tools, software and hardware, instead of purpose-built applications and devices.

2.2 Reference Architecture

The reference architecture, introduced in [5], is a general proof-of-concept control system platform for machine automation as an alternative to proprietary and specialized solutions. The platform is based on the ideas of real-time service orientation, introduced previously in this section, and emphasizes integrability, interoperability, maintainability and heterogeneity. Service orientation allows software components to be published and located locally or over a network.

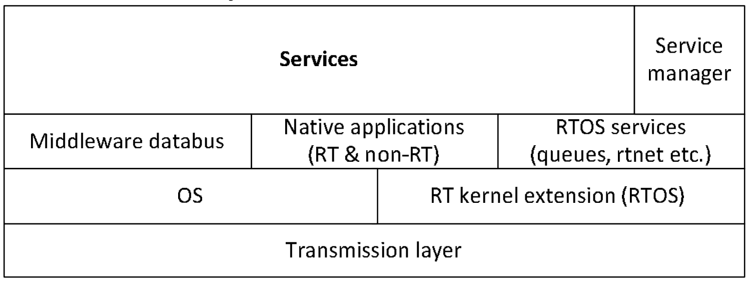
Services
Service manager
Middleware databus
Native applications (RT & non-RT)
RTOS services (queues, rtnet etc.)
OS
RT kernel extension (RTOS)
Transmission layer

Fig. 1. Layered architectural view of the reference architecture

In section 3 we will describe the actual implementation of the reference architecture for a remote handling scenario. A high-level layered view of the reference architecture is shown in Fig. 1. Key concepts of the architecture are services, communication & information sharing mechanisms, composition, and fault-tolerance. These are described next.

2.3 Concurrency Model & Real-Time Performance

The choice of a concurrency model for the architecture directly affects decoupling of the modules and management of real-time constraints. Options for the concurrency model include processes, threads or call-back functions [1]. Each solution has its own pros and cons for ease of development and inter-process communication. For a service-based architecture, the process-based model (services as processes) makes most sense, as it is the most decoupled alternative. This decoupling provided by processes has benefits, including the possibility to more easily manage services at runtime and improved robustness.

A service can be defined as an independently developed, deployed, managed, and maintained software implementation that directly represents business tasks or devices.

A service can be defined by a verb which describes the function it implements, e.g. “generate a trajectory”. Our implementation of services uses object orientation: service is an interface (virtual class) that has methods for starting, stopping, restarting etc. the service, which the service developer must implement. Services can use native applications and services provided by the operating system (e.g. APIs for communication).

2.4 Communication & Information Sharing

In order to communicate, components need some form of visibility or references between the communicating parties. However, this can lead to a tightly coupled system design that scales poorly. Examples of communication methods that impose coupling include sockets, remote method invocation and client-server model in general. A more decoupled solution is to use middleware based on the asynchronous publish/subscribe communication paradigm, which can be implemented as message-based like Java Message Service (JMS), or data-centric like Data Distribution Service (DDS)[1].

Another communication problem in distributed real-time systems is that networking can add unpredictable delays and unreliability to connections. Therefore we need to be able to set and monitor quality of service (QoS) parameters like reliability and how long the data is valid for each topic, so that the system can react appropriately if the QoS is compromised. QoS can be used to define if we want reliable sending (e.g. for commands) or just the most recent value as fast as possible (e.g. sensor measurements).

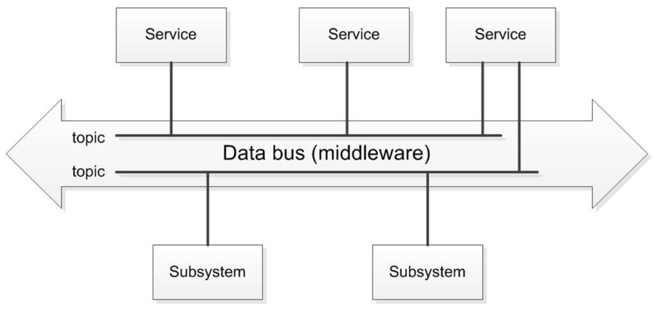

Fig. 2. Bus-based communication in SOA

The data-centric middleware can be used as a data bus between the services, as shown in Fig. 2: this is similar to the use of enterprise service bus (ESB) in enterprise SOAs. Another benefit of using a distributed middleware is a global data space where all data can be accessed; there is no central broker/repository that could act as a bottleneck or a single point of failure.

In an ideal situation we would have total location transparency for the services (no difference between accessing local and distributed services), but in order to achieve

[1] A standard maintained by Object Management Group, http://portals.omg.org/dds/.

optimal real-time performance, the architecture uses separate communication methods for local and networked communications, termed local service bus and global service bus. The reference architecture itself is not committed to any specific communication standard, but the implementation uses DDS middleware and the communication mechanisms provided by the real-time operating system (RTOS) Xenomai[2].

Local connection of services as components and the use of DDS as a data-bus for distributed communications combine the strengths of component and service approaches, and provides optimal real-time performance in both cases. DDS can be used on low-end embedded systems to read and send sensor information, whereas XML-based solutions would be too heavy, and would necessitate a separate solution.

- Global service bus: DDS was chosen since it implements asynchronous data-centric publish/subscribe model and provides QoS management, making it suitable for cyber-physical systems, which place a heavy emphasis on sending and receiving data.
- Local service bus: services can use RTOS message queues (an asynchronous “mailbox”) or shared memory for local real-time communication between two services. The queue-based local communication is similar to the component wiring approach used in component-based software engineering.

2.5 Composition

In complex systems, the number of internal components can easily grow to the range of hundreds or even thousands. Management of this many components or services can be complex and laborious if the framework-implementation of the architecture does not provide tools for this. Engineering of new applications from reusable components is supported by a repository of available components, configuration services to select and combine components, and run-time mechanisms that allow components to be dynamically changed.

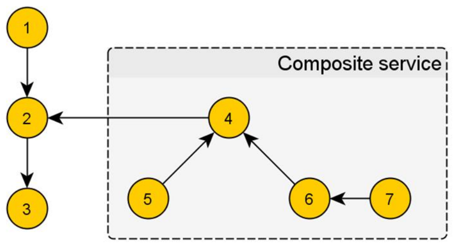

Fig. 3. Composite service (Key: circle denotes a service, arrow shows direction of data flow)

In the service-oriented architecture, higher level functionality can be implemented by creating composite services of the existing services, as shown in Fig. 3. Different means of implementing composition include programmatic, publish/subscribe, events and orchestration engine. Since our reference architecture is based on the pub-

[2] Real time Linux kernel extension and development framework, http://www.xenomai.org/.

lish/subscribe model, this is a natural match for the composition mechanism, and enables flexible implementation of composite services. Services can be chained locally and globally to form new composite services. A single service can be part of multiple composite services and used by multiple other services, which can reduce the level of unnecessary redundancy in the system.

A repository provides a way to document and list available services or components. For SOAs this can be done by writing an interface description and saving it in the repository. Service registries, on the other hand, provide runtime information for finding and binding services. In our proposed data-centric approach, based on the use of a data bus, the middleware can handle registration of new publishers, and match subscribers to the provided data topics.

Service composition and management at runtime is handled dynamically through a local service manager, which controls spawning of new services. This makes it possible to modify a service and restart it on-the-fly, enabling faster deployment process by updating only related services, instead of having to recompile the whole system after every reconfiguration or update.

2.6 Fault Tolerance

Fault tolerance is a key requirement for the architecture, as many cyber-physical systems perform safety-critical tasks. A fault in the control system may endanger human lives (either directly or indirectly), cause operational downtime or damage the environment or equipment. Service-orientation can support error confinement with the modular architecture, based on the decoupling provided by the service model, although the system still needs to implement error detection and recovery.

Because of the decoupled design, developers cannot make the presumption that other services are always available, and must take the situation into account in their application code so that the service will react if a dependency goes down, e.g. because of failure or manual shutdown. The error handling approach based on decoupling is similar to the one used in the Erlang programming language, which can be summarized as “let it crash” [7]. In the event of an error, the process is terminated, presuming it is not an exception that can be handled. This forces other services to react and do error recovery, including entering their safe state. The architecture can still be prone to error propagation, so the services should be made fail-silent if possible, making it easier to detect faults.

In order to implement error detection, the system can use a service manager to detect crashed services based on heartbeat signals or monitoring the use of resources like CPU and memory. Unresponsive behaviour or unexpected increase in CPU usage for a service can indicate a fault in the service, and may endanger real-time performance of other services and cause unexpected and potentially dangerous behaviour. The service manager restarts the unresponsive service, which will put the system temporarily into a safe state by forcing other services to do error handling, according to the “let it fail” approach. Key principle is writing loosely coupled services, by forcing the developer to consider situations where the dependency services are not available or timing constraints are violated.

3 Implementation for a Remote Handling System

In order to test the proposed data-centric real-time approach to service-orientation, we implemented a remote handling control system (RHCS) for automated teleoperation of an industrial robot Comau SMART NM45-2.0, based on the reference architecture described in the previous section. A basic remote handling scenario consists of an operator using the web-server based Operation Management System (OMS) to send movement commands to the equipment controller. Virtual reality software (IHA3D) is used to visualize the position and movements of the robot.

Services deployed on the equipment controller for the remote handling system implementation are shown in Fig. 4. Service descriptions, real-time task priorities and execution periods are listed in Table 1.

Table 1. List of services used in the remote handling control system.

Service name	Service description	Priority [0 .. 99]	Period [ms]
`Trajectory-Generator`	Generate a trajectory profile that the manipulator can follow from one point to another.	50	2
`C4G`	Interact with the low level control system of the manipulator.	91	2
`C4GJoint-DataPub`	Publish manipulator joint position data.	45	10
`OmsCom`	Read OMS commands and manipulator joint data; send commands to the trajectory generator to create new trajectories.	40	50
`Measuring`	Measure task execution time and jitter.	20	0.1

4 Evaluation of the Experiment

This section presents an evaluation of the problems and benefits of the proposed approach that could be observed with the implemented experimental system. The system is evaluated with the following criteria: reusability, interoperability, evolvability, real-time performance, fault tolerance, and ease of development. Dynamic composition performance depends greatly on the algorithm design [2] and it is not evaluated in this paper. Instead, a static composition is used.

Reused software includes `TrajectoryGenerator` service, `C4G` service and two subsystems (OMS and virtual reality). A service-based implementation avoids stovepipe system antipattern[3] as services are loosely coupled (no direct references to other services) and do not interfere with each other's namespaces etc., simplifying future reuse of services.

[3] http://sourcemaking.com/antipatterns/stovepipe-system

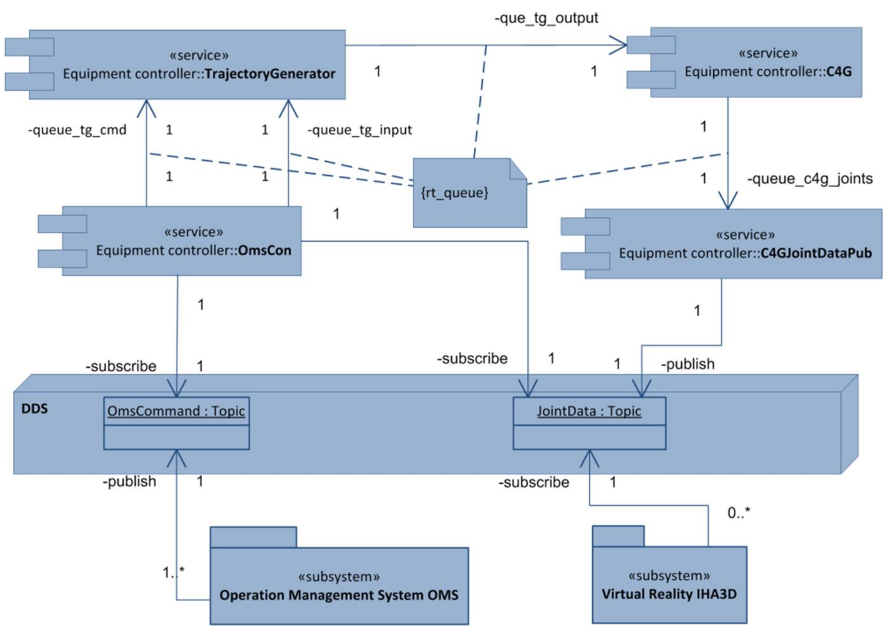

Fig. 4. Service deployment view for the system (Key: UML)

Interoperability of heterogeneous systems (machines and higher-level enterprise systems) is supported on any platform that has a compatible DDS implementation. DDS is available on several programming languages, therefore good programming language independence is provided. Interfaces to Web services, REST-based services and other communication platforms can be implemented with adapters.

Evolvability – the software must be able to accommodate new and changing requirements, including connections to unforeseen external sources. Ability to do this in the long term is especially important for industrial automation systems, because they have long expected lifetimes. This can be measured with evolvability, which describes the ability of software to accommodate future changes [8]. Performing a complete evolvability analysis is not reasonable in this context, so we focus on the changeability, extensibility and portability sub-characteristics:

- Changeability: Data typically has better consistency in the long run when compared to interfaces. However, if the data topics or queue configurations are changed or added, corresponding modifications must be implemented to both publishers and subscribers, but it is possible to provide extensions topics that provide the new or changed data, thus retaining compatibility with old implementations.
- Extensibility: New topics or functionality in the form of services can be added on-the-fly, without shutting down and recompiling the whole system. The runtime composition can be managed with the service manager, which can also be used to lazily launch necessary services (service chains).
- Portability is limited if RTOS-specific features like real-time queues are used.

Real-time performance – we analysed system performance by measuring cycle durations for a real-time task first unloaded and then running a full remote handling system with a script generating artificial CPU, network & disk loads. The real-time measuring task was executed 10000 times with 100 µs period on a 3.4 GHz Pentium 4 CPU. The measured latencies are shown in Fig. 5. Although standard deviation of cycle duration has increased from 172 ns to 410 ns in the heavily loaded system, graphs show highly deterministic behaviour in both cases. Performance of the DDS middleware in embedded real-time systems has been evaluated e.g. by Xiong et al. in [11].

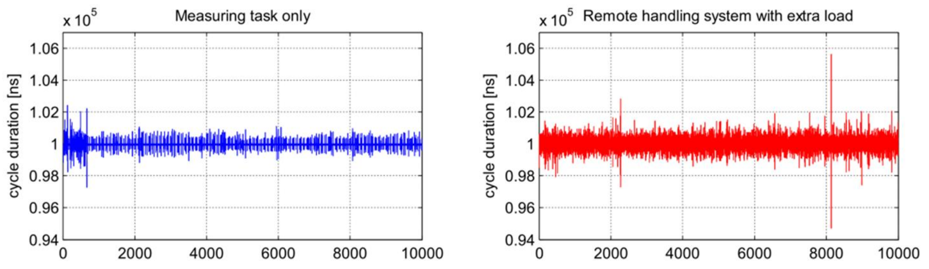

Fig. 5. Cycle durations for single task vs. remote handling system with extra load

Fault tolerance – the service manager can detect if services use more system resources than reserved at start-up, and force a restart. Other services need to react according to the "let it crash" error handling approach. After the services have been restarted, normal operation can be resumed if the fault was transient. A leaky bucket counter or an escalating retry timer can be used to distinguish transient faults from permanent ones.

An example case of error handling: the `TrajectoryGenerator` service is killed in the middle of running a trajectory to the `C4G` service, which controls the robot. `C4G` service detects that there is no new data available, and stops the movement of the robot, by ramping down the power in a controlled and safe fashion. Normal operation can be resumed when the `TrajectoryGenerator` is restarted.

Cost-efficiency & ease of development: the service-model is an intuitive approach for developers, as services can be interfaces to devices or related to tasks that must be accomplished. Linux-based development offers a variety of tools & drivers, reducing need for self-developed or proprietary choices. Communication configurations (for local queues) are currently hardcoded, so managing a large number of local communications becomes cumbersome, although the service manager can be used to start services. The local service communications should be standardized and details moved to external configuration files that could also be managed with tools to simplify management and reduce local coupling between services.

5 Conclusions

A dynamic module system based on services or components is necessary to manage complexity of embedded and distributed control systems. The module system should abstract the communications between modules, and provide tools for managing and deploying the configurations in order to improve software reusability and simplify development process, maintenance, and integration of new devices to the system.

In this paper we have presented our design concept for a service-based software architecture. Our proposed approach adapts the SOA paradigm with data-centric design, based on topic-based publish/subscribe middleware and RTOS. The experimental implementation of the architecture demonstrates integration of heterogeneous subsystems with the service-based control system through a scalable middleware-based data bus. The control system is based on an open RTOS and has deterministic real-time capabilities. Although all composition features in the prototype are not fully implemented, it provides contribution by testing the data-centric approach to implementing RTSOA.

References

1. Calisi, D., Censi, A., Iocchi, L., & Nardi, D.: Design choices for modular and flexible robotic software development: the OpenRDK viewpoint. Journal of Software Engineering for Robotics, 3(1), pp. 13-27. (2012)
2. Tsai, W., Lee, Y.-H., Cao, Z., Chen, Y., & Xiao, B.: RTSOA: Real-Time Service-Oriented Architecture. Proceedings of the 2nd IEEE International Symposium on Service-Oriented System Engineering (SOSE'06), pp. 49-56. IEEE (2006)
3. Cucinotta, T., Mancina, A., Anastasi, G., Lipari, G., Mangeruca, L., Checcozzo, R., et al.: A Real-Time Service-Oriented Architecture for Industrial Automation. Industrial Informatics, IEEE Transactions on , 5(3), pp. 267-277. IEEE (2009)
4. Moussa, H., Gao, T., Yen, I.-L., Bastani, F., & Jeng, J.-J.: Toward effective service composition for real-time SOA-based systems. Service oriented computing and applications, 4(Special Issue: RTSOAA), pp. 17-31. Springer (2010)
5. Hahto, A., Rasi, T., Mattila, J., & Koskimies, K.: Service-oriented architecture for embedded machine control. International Conference on Service-Oriented Computing and Applications. IEEE (2011)
6. Machado, A., & Ferraz, C.: Guidelines for performance evaluation of web services. WebMedia '05 Proc. of the 11th Brazilian Symp. on Multimedia and the web, pp. 1-10. ACM (2005)
7. Armstrong, J.: Making reliable distributed systems in the presence of software errors. Dissertation, Stockholm, Sweden: Royal Institute of Technology (2003)
8. Pei-Breivold, H., Crnkovic, I., & Larsson, M.: A systematic review of software architecture evolution research. Inf. and Software Technol., 54(1), pp. 16-40. Elsevier (2012)
9. Panahi, M., Nie, W., & Lin, K.-J.: A Framework for real-time service-oriented architecture. Commerce and Enterp. Comput., 2009 IEEE Conf. on, pp. 460-467. IEEE (2009)
10. Garces-Erice, L.: Building an Enterprise Service Bus for Real-Time SOA: A Messaging Middleware Stack. Computer Software and Applications Conference COMPSAC '09, 33rd Annual IEEE International, pp. 79-84. IEEE (2009)
11. Xiong, M., Parsons, J., Edmondson, J., Nguyen, H., & Schmidt, D. C.: Evaluating the performance of publish/subscribe platforms for information management in distributed real-time and embedded systems. (2011)