A Runtime Resource-Management Framework for Embedded Service-Oriented Systems

Peter Newman and Gerald Kotonya
School of Computing and Communications
Lancaster University
Lancaster, LA1 4WA, UK
{p.newman,gerald}@comp.lancs.ac.uk

Abstract—Service-Oriented Computing (SOC) has rapidly progressed in recent years, with examples appearing in numerous application areas including business processes, high-performance computing, web-based services, and embedded systems. In the embedded systems domain, ubiquitous devices, context-aware systems, and sensor networks are increasingly being used in a plethora of areas. However, because of the nature of their operating environment, service-oriented embedded systems (SOeS) pose unique challenges to quality management. SOeS have to contend not only with the quality of services from providers, but also with the constrained system resources in their operating environment. SOeS need to efficiently utilise available system resources by dynamically adapting themselves to changes in their runtime environment. This paper discusses current SOC resource-management initiatives, setting out the challenges that SOeS systems must overcome, and proposes a pluggable, runtime resource-management framework for SOeS.

Keywords—Emergent Service Properties; Runtime Architecture; Resource-Management; Service-oriented Systems

I. INTRODUCTION

The service-oriented model of software deployment offers significant benefits over the traditional model of software deployment as a product, including reduced capital investment, dynamic integration and rapid deployment of platform and network-independent systems [1]. Service-oriented architecture (SOA) provides the conceptual framework for realising this vision by allowing software systems to be dynamically composed and reconfigured using services discoverable on the network. This dynamic model of software deployment is particularly attractive for embedded systems as it offers a potential solution to several design challenges faced by embedded system designers [2]. Embedded systems have a limited capacity to carry programs that handle all situations; as such, they tend to be bespoke for specific tasks. Resource restrictions in embedded systems mean that small changes in their operating environment are likely to have an adverse effect on system quality. Additionally, many embedded systems operate in critical application domains where modifications often need to be made without stopping the system [3].

As the nature of embedded service-oriented applications continues to vary and the demands on them grow, features such as dependability and adaptability to different runtime environments are becoming increasingly important consumer quality considerations. Embedded service-oriented system quality is not just a function of the quality of a provided service, but the interdependencies between services, competing quality requirements, resource constraints of the runtime environment and network outages. This emergent Quality of Service (QoS) is difficult to anticipate before deployment as the changes in QoS are dynamic and influenced by a myriad of factors. Emergent QoS is even more significant within the embedded domain as resource contention between various services becomes more apparent; this is due to the naturally constrained resources of the embedded platform.

There are a number of research initiatives currently investigating effective ways to monitor embedded-system resources at runtime and a sizable number that propose the management of resources through the use of pre-determined degraded QoS policies; however they are largely based on static quality properties [4], [5]. We believe that a more effective solution lies in a dynamic runtime resource-management framework. This paper proposes a runtime resource-management framework that combines resource monitoring with task-flow management and dynamic orchestration to optimize system performance.

II. CHALLENGES

Current approaches to ensure the runtime QoS in embedded systems are insufficient; more specifically, there are a lack of research initiatives investigating the negative impact resource contention can have on resource constrained systems, and the dynamic reconfiguration of executing services to mitigate this resource contention.

Current quality assurance approaches have the following limitations:

- Poor support for runtime quality assurance. Existing service monitoring initiatives are largely manual approaches, which focus on static service properties, and lack the ability to effectively recover from a Service Level Agreement (SLA) violation or service failure. Real-time service contracts between components, as
an important aspect of QoS considerations, add a new dimension to the development and validation of component-based embedded systems [6]. In contrast, the static monitoring approaches currently employed provide poor support for detecting and responding to emergent runtime quality problems in the service execution environment.

- **Poor support for resource-constrained systems.** Ensuring quality is particularly problematic for service-oriented systems which operate in resource-constrained environments [7]; not only must a service provide an acceptable QoS, but it must be capable of being integrated within the resource constraints of the service consumer. Resource contention between services is more significant within resource-constrained orchestration environments as services must contend for a limited pool of resources.

- **Static service orchestration.** Existing quality assurance approaches for service-oriented systems are based on static service orchestration; this does not take into account the status of the service and changes in system resources. This can result in an application that is inefficient and performs poorly [8].

### III. RELATED WORK

Much work has been done with regards to resource monitoring/management within the embedded domain. The work of Huber et al. [9] for example presents a resource management framework which provides ‘static’ resource guarantees for Distributed Application Subsystems (DASs) that have higher dependability requirements whilst providing efficient resource utilisation for less critical DASs. This framework allows for the monitoring of the resources of the environment (at runtime) and to dynamically allocate them should QoS issues arise. This paper does not however address the pre-emptive degradation of service quality; by this we mean that the QoS levels for each of the DASs is decided during the design phase of the component. As such, the DASs components must be willing to yield their resources via lowering their QoS level (based on predefined levels of service degradation).

As our work is influenced by the SOA paradigm, it is important to investigate current methods of service monitoring/management. One such method of service management is proposed by Robinson and Kotonya [8]. They presented an architecture that provides a self-managing mechanism via Service monitoring, Negotiation, Forecasting and Vendor reputation for ensuring runtime quality in service-oriented systems. The architecture describes a quality framework, developed using JINI SOA technology but also capable of supporting other SOAs such as web-services. The focus of the research was aimed at desktop computational platforms rather than embedded resource-constrained systems. The paper does not address issues specifically related to emergent-QoS; instead it addresses the failings of the services in isolation, monitoring their individual QoS metrics without including the service within the context of the orchestration environment.

Moser et al. [10] deal with the monitoring and runtime configuration of services. More specifically, the paper presents a system named VieDAME; this allows monitoring of BPEL processing according to a specified QoS characteristic-set and the replacement of existing services for semantically and syntactically similar services using a specified replacement policy. It is mentioned that current BPEL engines do not have the capability to monitor processes on a real-time basis; as such VieDAME is meant to address this lack of real-time monitoring. Although this paper addresses issues with monitoring of services at runtime, it only addresses using web services and not other forms of SOA.

Because our work deals with task-flows, it is important to look at current methods of task-flow ordering and management. The work of Dyachuk and Deters [11] proposes to schedule service requests in order to improve the overall Composite Web Service (CWS) performance in overload situations. Specifically, they discuss how using different task flow policies can achieve better performance. The novel Augmented Least Work Remaining (ALWKR) scheduling policy proposed in this paper is reported to achieve better response times than that of other such scheduling policies (such as FIFO and SJF). ALWKR is similar to the Least Work Remaining (LWKR) scheduling policy, however it uses the workflow topology when scheduling; effectively, this allows services to be called in a concurrent fashion rather than just sequentially.

### IV. PLUGGABLE RESOURCE MANAGEMENT FRAMEWORK

In order to fulfil the challenges mentioned in Section II, this paper proposes a resource-aware framework for use within the service-oriented embedded domain. The way in which the framework reconfigures the service orchestration is dictated via a selection of custom service policies; as such, the framework is pluggable with new behaviour.

Figure 1 illustrates a high-level view of the proposed framework architecture; it describes how the framework interacts with the orchestration host. The SOA operates on the orchestration platform, consuming system resources; the Resource Monitor uses the API of the platform to monitor this consumption. Once specified resource levels have been reached, the Taskflow Manager will trigger changes within the SOA orchestration, reconfiguring the active services in a way dictated by its service policy. Not only are the services themselves altered, but the way in which they execute can be reconfigured; their ordering and as such dependencies can be changed, fundamentally altering the flow and structure of the SOA Orchestration. The Taskflow Manager composes
the services into a graph of tasks to be executed; this graph represents the orchestration of services and thus is the SOA orchestration itself. As such, the Taskflow Manager could be considered a component of the SOA orchestration, although only in an implicit sense; this is represented in Figure 1 as a dashed box surrounding SOA Orchestration and Taskflow Manager.

Figure 2 describes the architecture of the framework in more detail. The Framework Controller component acts as a facade, managing the components of the framework and generating the Service Manager based on the supplied XML configuration file. All other components interact with each other through the Framework Controller component using a combination of synchronous and asynchronous (events) calls. The Resource Monitor component is responsible for monitoring available system resources; these resources include CPU load, memory, network usage etc. Attached to the Resource Monitor are a set of resource readers; they are responsible for monitoring actual systems resources. These readers are bespoke for Windows NT and Unix-based platforms; this is because both provide different methods to monitor system resource levels, with Windows providing an API (windows.h) whilst Unix-based platforms typically use the proc file. Once undesired levels of resource usage have been reached, the Framework Controller component will trigger the execution of a service policy. Service policies are implemented using a scripting language (such as Lua\(^1\)) so that the integration of new service policies is possible, allowing for pluggable behaviour.

The service policies specify the behaviour of the framework and how services are managed during resource contention; interaction with the framework is achieved via the Service Policy Manager component. The Service Policy Manager component allows for Lua scripts to issue events to the framework and maintain state between multiple invocations of a script. Within the maintained state, global variables can be stored for use in future invocations of the service policy. Actual service interaction is delegated to the Service Manager component; this component is extensible, allowing for new service management behaviour to be added. The Service Manager is responsible for maintaining currently active services and handles the execution of these services; it obtains these services via the Service Gateway class. Because of the numerous protocols for obtaining services over a network (such as SOAP and Jini), the Service Gateway class has also been made extensible, allowing the developer to obtain services from a bespoke source. Because the Service Manager holds the set of services for the SOA application and is responsible for the invocation of these services, it is considered to be the SOA application itself; the dashed outline surrounding both the Service Manager and Taskflow Manager illustrates this in both Figure 1 and 2.

During the evaluation of our framework, we plan to use service doping mechanisms to alter the QoS provided to the consumer devices at runtime. Because neither the real-world or simulated system case-studies may result naturally in a system with overloaded resources, we will also be artificially constraining resources within the orchestration platform to simulate periods of severe resource contention. These doping mechanisms will enable us to explore different QoS scenarios, highlighting how the framework supports the consumer in maintaining satisfactory quality levels while using the application.

V. EARLY RESULTS

At present, resource usage is monitored and service policies are invoked when resource usage has reached undesirable levels. Figure 3 illustrates a UI developed to evaluate the performance of our framework; resource usage is displayed graphically at real-time, showing changes in the runtime environment as well as the current moving average. Although a number of different resource metrics can be monitored, in this instance; CPU load, memory load and free physical memory are being monitored. The screen-dump presented shows a recent increase in the amount of physical memory available to the system; this is confirmed by the memory load value, which shows a marked decrease.

We have thus far tested the framework on a small SOA system. To simulate the system working within a resource-constrained environment, we have doped the resources of the desktop platform. We created a Service Manager and Service Policy to allow for the hibernation of services; this is where services are removed from memory when not required and rebound when needed during runtime. The result of such hibernation is illustrated in Figure 3.

Initial tests yield promising results, as the memory footprint of the SOA does show a marked decrease during

---

\(^1\)http://www.lua.org/
the hibernation of services, although further testing must be conducting on larger systems to determine any tangible benefit.

VI. Final Remarks

Thus far, we have implemented a prototype system utilising Sun Microsystems’ Jini\(^2\) platform; initial results show promise. The service policies we have created do relieve some resource contention, although through the development of more service policies better results may be obtained via different service strategies. Future work includes; runtime environmental modification of the service consumer should SLA violations be detected, allowing the consumer to monitor service quality properties at runtime, and the automation of service discovery and invocation (should a service require replacement).

REFERENCES


