Annales Univ. Sci. Budapest., Sect. Comp. 30 (2009) 191-200

NODE-BASED ARCHITECTURE FOR LIGHTWEIGHT MIDDLEWARE

V.-M. Hartikainen, M. Vulli and H.-M. Järvinen

(Tampere, Finland)

Abstract. We propose a simple middleware architecture for mobile devices. The architecture is based on nodes and streams. In a REST-like way, the nodes only provide standard operations for control. Several middleware solutions have been created for developing distributed software. However, the existing solutions do not suit well for use in embedded or mobile devices. By specifically targeting the design of the middleware to mobile domain we can focus on the most relevant issues and still keep the design of the infrastructure simple, thus allowing the components to be implemented in software or hardware. The architecture proposed changes the programming paradigm, making it more suitable for component-oriented development.

1. Introduction

Service-based architecture promises a lot of things. One of the biggest promises is that building of big systems can be made simpler by dividing them into independent components. This is already happening in the world of enterprise computing with Service-Oriented Architecture and Component technologies. In our opinion, the world of mobile computing could benefit from a service-based architecture, where the infrastructure is targeted to mobile devices. The infrastructure should be loosely coupled, reflective, flexible, dynamic, lightweight, simple for developers to understand and offer possibility to implement the components in hardware or software.

In the mobile and embedded world we just cannot use the Web Service technologies to connect the software components inside the device due to the constraints of the devices (performance, footprint, etc.), but we can learn from their best practices. Component technologies (such as Corba, DCOM and J2EE), even in their light-weight implementations, have not gained foothold in the mobile devices. There are many reasons for this. We think that standard middleware solutions have not yet been successful in the mobile world, because they are not designed for embedded devices. Therefore they are heavy-weight, complex and too generic. The standard technologies do not really support the possibility for implementing a component in hardware and therefore require additional software wrapping of hardware.

One of the most valuable lessons that we have learned from variety of distributed systems is that tight coupling eventually leads to integration problems. When abstraction level has been raised in developing distributed systems, the coupling between participants has loosened. For example, Corba over proprietary binary protocols, Web Services over Corba, REST over Web Services.

Decreased coupling allows the nodes to communicate with each other when they are using the subset of messages that both understand. When trying to use other messages we still may get something useful done. However, the main point here is that extending node's interfaces with new messages and messages with new parameters in a really loosely-coupled environment never brakes anything and therefore, it is easier to add compatible components to the system or upgrade current components with new features.

This paper presents our vision for lightweight middleware, targeted for mobile devices, that provides a service-based architecture. The target hardware is above the smallest embedded controllers, but far below desktop computer performance. For example next-generation hand-held products such as video-phones, portable communicators, PDAs or other consumer products like set-top boxes and digital video cameras. These devices typically have 32-bit RISC processors with few hundred MIPS computing capability.

2. Node-based architecture

In this architecture we have nodes and streams. The data is transported in streams between the nodes and the nodes operate on the streams. Nodes are controlled with a fixed set of standard operations. Application developer uses Unified API to control the system. The ideology of the approach is quite close to the ideology in the REST-style of implementing Web Services [1]. Our architecture resembles architecture of GStreamer-framework. There are also similarities

between our approach and traditional Unix approach, where different kind of resources (devices, files, processes) can be accessed with standard file operations.

2.1. Unified API

Service discovery has been traditionally initiated by a client. The client looks up a server and then starts to send requests to the server. In a multi-tier architecture, a single component can operate both as a client and a server (or both). Traditional architecture may easily lead to situations where both the client and the server are tightly coupled to each other. Since nodes are usually connected also to other nodes, we may end up having a long chain of dependencies. An alternative and a more dynamic approach is to let an external party (control) to connect the nodes to each other based on capabilities of the nodes. For example, when user wants to play a video, control looks up a video source and then video output. After finding the nodes the control connects the nodes with a stream and then starts the stream. Figure 1 illustrates structure of the video player application.

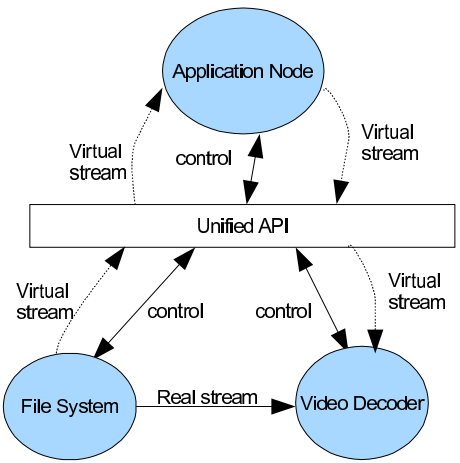

Figure 1. Structure of simple video player

This means that a node programmer sees only Unified API and streams. The middleware takes care of the rest. Developers do not need to know that, behind the API, services are provided by several nodes. The infrastructure is made in such a way, that from node developer point of view, the node only communicates with Unified API. The infrastructure transparently relays the communication to

correct recipient. Application programmer does not need to know if the nodes are local or distributed. The middleware hides the differences of local and remote nodes.

Application developer needs to define what streams he wants to use and what operations the application wants the system to perform to the stream. For example if an application wants to play a MP3-file, the application first asks from the middleware a stream with a file path. The middleware finds a node that can produce a stream from the file path. The middleware then asks the node to produce the stream. After that the application asks the middleware to modify the state of the stream to played state. Middleware then looks up a node or a chain of nodes that do the necessary operations on the stream (including possibly decoding and playback) and connects all the nodes needed to the stream. Finally, the application asks the middleware to start the stream. Figure 2 illustrates how a video stream could be played.

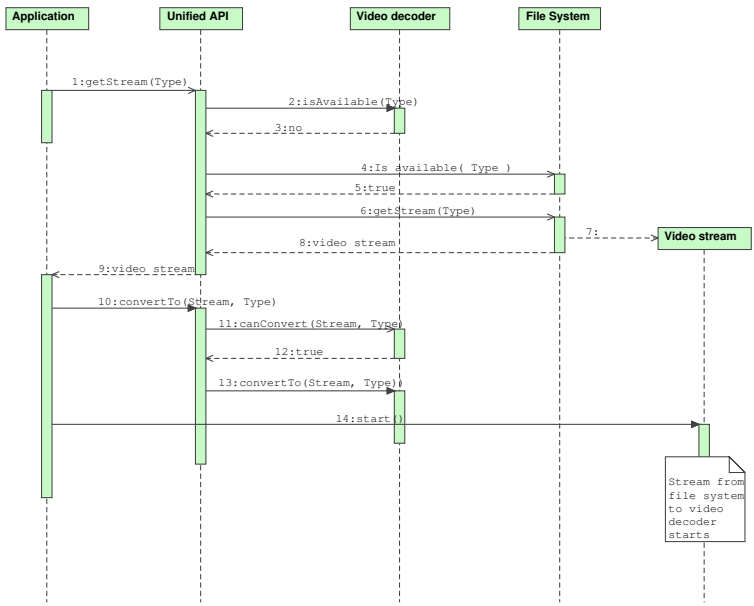

Figure 2. Opening a video stream

2.2. Nodes

A node is the basic component of the system. It is an entity, which is able to perform some operation. For example, there might exist a node which would take an audio stream as input, and then output it as a compressed audio stream. Nodes can be implemented with software or hardware. In a networked device a node can be remote or local.

2.3. Streams

A stream is data transport between nodes. A stream is created by a node that can produce some data on the middleware's request. The middleware requests a stream from a node, when another node requests to receive or manipulate a stream. For example, when application wishes to receive a stream. Although, a stream connects two nodes, neither node is responsible for opening the stream. Hence, the middleware decides which nodes to connect, and the middleware connects them with a stream in the most appropriate way.

The type and the state of stream is described by properties. The properties are divided to property classes. The properties and property classes form a tree structure, which reminds of a directory structure in a file system. For example, a property describing a JPEG-image stream would be `/media/image/jpeg`.

Properties may contain attributes. An attribute is a name and value pair describing the property. For example, the above-mentioned property for jpeg-image stream could have integer-valued attributes `/media/image/jpeg/width` and `/media/image/jpeg/height` describing the width and height of the image, respectively.

Nodes use properties to describe what kinds of streams they can consume and produce. Middleware uses properties to find what nodes can be connected to each other.

2.4. Control

The nodes are controlled using a predefined interface. All nodes implement the same interface and the nodes cannot provide any additional operations. Since purpose of nodes is to manipulate streams (create, show, modify, etc.) the operations are mostly related to manipulating streams. There are operations for controlling the flow of stream (start, stop, hand-over) and operations for controlling properties of the stream. There are operations for checking what kind of streams the node is able to receive and what the node is able to produce out of those streams.

3. Lightweight middleware

It is quite clear that for implementing the architecture proposed, a middleware is needed. The middleware needs to be so light-weighted that its services can be implemented in hardware. Whether the service is actually an independent unit on a chip or a software service offered by a CPU, should not really matter for application developers.

In a typical middleware-based distributed solution, the software stack contains five layers below the application. Figure 3 shows a software stack for typical distributed software systems. Transport layer is responsible for transferring data from component to component. The messaging layer transforms high level messages (such as remote procedure calls) to a communication protocol that can be sent using the transport layer. The description layer describes services. The discovery tracks service providers so that service consumers can locate and utilize services provided.

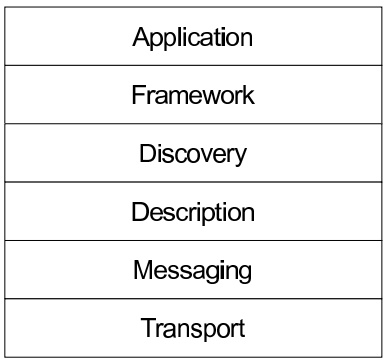

Figure 3. Typical distributed application software stack

In the proposed architecture the application developer does not really need to use the discovery layer, since the application node only needs to communicate with Unified API. However, there is really no good reason to deny access to discovery services, although normally they would not be needed. Since we allow usage of discovery services also for nodes, it is possible to explicitly compose a higher-level service from other nodes. Since it is possible to explicitly state which nodes are used, we can also know the characteristics of these nodes beforehand, thus making it easier to create deterministically behaving flows for time-critical situations.

Since all nodes are controlled using the same interface, the description layer does not need to describe the interface for nodes. Instead it is responsibility of the description layer to know the types of streams nodes can produce and consume, and what kind of transformations the nodes can do to streams. For software nodes and intelligent hardware nodes the features are queried dynamically during run time. The middleware needs to store the same information for those (simple) hardware nodes that cannot answer queries of this nature during run time.

Messaging needs to use protocol that is simple enough so that it can be easily implemented in hardware. Here we are not really interested in the problems and solutions of transport level, since it is mainly a concern of hardware and networking people. So from our view point, the software stack should be transport agnostic and the underlying transport may be TCP/IP, some chip-internal transport on a network-on-chip device, or some other form of communication.

In addition, the middleware has resource management responsibilities and needs to be aware of capacity of hardware, networking costs, etc. so that quality-of-service type of needs can be met.

With middleware, it is possible to use multiple programming languages. We can use C to implement the performance critical components and the middleware itself. For building applications one can then use a higher-level language and thus enjoy the greater productivity of higher-level languages. However, since the proposed middleware is targeted to be light-weight, the C-language has to have the top priority.

4. Related work

Schmidt et al. collected the challenges for distributed real-time and embedded systems. If all the challenges could be answered, the payoff would be a reusable middleware that simplifies the development of large distributed real-time and embedded systems. There is a demand for end-to-end QoS (Quality of Service) since usability of the resulting products is dependent on properties of the whole system. The solutions need to be adaptive and reflective, so that they can handle both variability and control. The middleware itself is not enough to deliver the capabilities envisioned for next generation embedded systems. Advances in system engineering approaches and tools are also needed [2].

Capra et al. explored reflection in mobile computing middleware. Current generation of mainstream middleware is heavy-weighted, monolithic and inflexible. Current middleware is not well suited for mobile environments where resources are scarce. Mobile middleware needs to be context-aware and needs to

cope with heterogeneous platforms. Reflection is one possible solution to challenges of mobile middleware, since it allows more configurable and reconfigurable middleware. University Collage in London has project CARISMA, where reflection is used to support dynamic adaptation of middleware behavior to changes in context. On the other hand, Lancaster University's ReMMoC project uses reflection to accommodate heterogeneity requirements imposed by both applications and underlying device platforms. Drawbacks of the reflective approach are in performance, integrity and security. The reflection is not enough, also other mobile code paradigms are needed for having a good level of dynamism and flexibility in the middleware. [3]

Batista et al. used a scripting language called Lua to dynamically interconnect component-based applications. Architecture is composed of CORBA components, LuaORB and Lua-language. Lua is a interpreted, dynamically-typed language and it offers several reflective facilities. LuaOrb is a binding between the language and CORBA. LuaOrb allows dynamic installation of implementations written in Lua. LuaOrb exploits the dynamic features of Lua to access CORBA objects like any other Lua objects at runtime [4].

Michi Henning analyzed what can be learned from CORBA's mistakes. The main reasons for the fall of CORBA according to Henning are complexity, insufficient features and interoperability problems. The lessons learned are: standards should be followed more strictly, standard should not be approved without a reference implementation and standards should be tested in realistic scale projects before being accepted [5].

Mungee, Surendran, and Schmidt described an implementation of OMG audio/video streaming model, which is based on TAO, a real-time CORBA ORB. CORBA IIOP is used for control and stream establishment, while streams are allowed to use more efficient transports, like ATM, TCP, or UDP because they do not go trough the ORB. Separating control and data is necessary to achieve high performance. However, this CORBA-based solution is not lightweight enough for embedded devices [6].

As discussed in this article, we have mentioned SOA and Web Services as examples for our architecture. Using Web Services from mobile devices is possible with JSR-172 J2ME Web Services API, which adds support for using Web Services to mobile Java. This allows usage of Web Services with SOAP communication from mobile devices. JSR-172 operates on higher-level and uses Java. The services are higher granularity and since the services are accessed through network, lower latencies are accepted. It suites well for integration high-end mobile devices to external Web Service world. JSR-172 is not a performance-oriented API suitable for low-level integration inside the device, which is the main target of the architecture presented in this article [7].

5. Conclusions and future work

In this paper, we presented a simple architecture for lightweight middleware, which in our view is capable of answering some of the challenges in the mobile domain.

Unified API offers unique way for developing component-based applications. It eases developers work a lot, but on the other hand makes developing the actual middleware quite challenging. Especially, since the target is to have lightweight middleware suitable for embedded and mobile devices. We can really only say if the idea works or not after we have a running prototype.

One of our main ideas is to control the nodes only using standard operations. This brings in flexibility and loose coupling as the interfaces always remain the same. Deciding the actual set of standard operations is a very important task and it will probably require some prototyping before the set of operations is most appropriate.

Reflection on the system is achieved through properties that describe the streams' and the nodes' capabilities on operating on the streams. There is a lot of work in designing the property system. We need to have a rich model how to describe the streams and currently our model is not very specific. Since streams are controlled through the properties, we need to have standard properties for well known types of streams. We need to have a lot of conventions and standards for describing the streams. We need guidelines on how the property hierarchy is formed. Otherwise, the control through just using standard operations will not be possible.

References

[1] **Fielding R.T.,** *Architectural styles and the design of network-based software Architectures,* PhD Thesis, University of California Irvine, Information and Computer Science, 2000.

[2] **Schmidt D.C. and Gokhale A.,** Middleware R&D Challenges for distributed real-time and embedded systems, *SIGBED Review,* **1** (1) (2004).

[3] **Capra L., Blair G. S., Mascolo C., Emmerich W. and Grace P.,** Exploiting reflection in mobile computing middleware, *ACM SIGMOBILE Mobile Computing and Communications Review,* **6** (4) (2002), ACM Press, 34-44.

[4] **Batista T. and Rodriguez N.,** Using a scripting language to dynamically interconnect component-based applications, *6th Brazilian Symposium on Programming Languages,* 2002, 180-194.

[5] **Henning M.,** The rise and fall of CORBA, *ACM Queue*, **4**(5) (2006), http://www.acmqueue.com/modules.php?name=Content&pa=showpage&pid=396

[6] **Mungee S., Surendran N. and Schmidt D. C.,** The design and performance of a CORBA audio/video streaming service, *HICSS-32. Proceedings of the 32nd Annual Hawaii International Conference on System Sciences*, 1999.

[7] **Java Community Process**, *JSR-172: J2MET Web Services Specification*, http://www.jcp.org/en/jsr/detail?id=172, referred at 17.5.2007.

V.-M. Hartikainen, M. Vulli and H.-M. Järvinen
Institute of Software Systems
Tampere University of Technology
P.O. BOX 553
FIN-33101 Tampere, Finland
{vesa-matti.hartikainen, mikko.vulli, hannu-matti.jarvinen}@tut.fi