ActiveLink
An embedded systems solution for generic cross-platform communication

Bernard M. Venemans[a], Michel Chaudron[a], and Harro S. Jacobs[b]

[a] Eindhoven University of Technology, Department of Computing Science,
P.O. Box 513, 5600 MB Eindhoven, The Netherlands, e-mail: m.r.v.chaudron@tue.nl

[b] CMG Eindhoven B.V. - Sector Trade, Transport & Industry, Luchthavenweg 57
P.O. Box 7089, 5605 JB Eindhoven, The Netherlands, e-mail: tswe@cmg.nl

Abstract - **A trend in automation is the increasing connectivity of devices and applications in order to provide new types of functionality. To combine functionality, it is necessary that devices can communicate with each other. In recent years, we have seen the emergence of a special type of software, called *middleware*, that aims at inter-application communication.**

In this paper we present ActiveLink: a solution for cross-platform communication between embedded systems. ActiveLink is lightweight, easily portable, and supports many communication protocols. ActiveLink has been developed by CMG, supported with research from the Eindhoven University of Technology [1].

We discuss ActiveLink in the context of the requirement for embedded systems and compare its features to that of mainstream middleware.

Keywords - **middleware; cross-platform communication; embedded software; portability.**

I. INTRODUCTION

As more and more devices are being embedded with software, it becomes possible to connect these devices to combine their functionality. This is already happening in some domains, such as consumer electronics, and is very likely to become more common in the near future. Communication between devices allows these systems to use each other's features. In this way, devices can specialize in their most important quality. Your television is specialized in graphical display and your heating system in heating. Connecting these two systems enables the heating system to show the current temperature in your living room via your television. In addition, you can use your remote control to change the room temperature or to program a temperature schedule for the coming week. The latter possibility requires an advanced displaying device, which, if integrated with the heating system, would have a dramatic impact on price. The costs to realize connectivity between devices are dropping sharply. Consequently, increased connectivity can strongly improve the ease-of-use, with only little impact on price.

The wish to connect devices calls for techniques for composing applications. Middleware is emerging as a key technology for constructing and integrating distributed applications. Middleware forms a layer between the application and the network, and allows applications to use functionality that resides on remote systems. Important middleware standards are OMG's Common Object Request Broker Architecture (CORBA), Microsoft's Distributed Component Object Model (DCOM), and Sun's Java Remote Method Invocation (Java/RMI). Examples of more recent middleware systems are Sun's Jini and Microsoft's .NET.

CMG is acting in the domain of automated testing, and recently extended its tool-set to test embedded software [2]. For this purpose, a host system running the software that controls the testing (typically a PC) and an embedded system running the software to be tested had to be connected. The middleware standards mentioned above are unsuitable to realize this connectivity, because they have too much impact on the embedded system. As no suitable, generic solutions were available, CMG developed ActiveLink: a highly portable small-sized utility for generic cross-platform communication.

In this paper we present ActiveLink as a solution for middleware for embedded systems. We discuss some key architectural issues that are motivated from requirements from the domain of embedded systems.

II. MIDDLEWARE REQUIREMENTS

Middleware is software that facilitates inter-connectivity between applications in distributed platforms. To this end, middleware provides a set of services to applications running on separate systems across a network. In Figure 1, the position of middleware in the OSI 7-layer reference model is depicted [3].

An important goal of middleware is to simplify the programming of distributed systems. One of the ways in which it tries to achieve this is to make distribution of the system transparent to the applications.

To apply middleware in embedded systems, some specific requirements hold.

A. *Platform support*

A characteristic of the embedded systems domain is its large diversity of processor architectures (e.g. x86, MIPS, ARM, SPARC, TriMedia) and operating systems (e.g. Windows, pSOS, VxWorks, Solaris). Middleware for embedded systems should be applicable to a large subset of these platforms. Hence platform-specific assumptions must be avoided.

B. *Protocol support*

Embedded systems typically communicate using low-level protocols at the physical boundaries of devices. A large variety of such protocols exists (e.g. RS-232, USB, and TCP/IP) and sometimes even dedicated protocols are used. Middleware for embedded systems should support a large subset of these protocols. Hence, it should not make any assumptions on the protocol and be easily extensible with new protocols.

C. *Size*

The available resources of embedded systems, such as processing power and memory size, varies considerably. To be applicable in this wide range of embedded systems, the memory footprint of the middleware should not exceed 25% of the total memory space. Given an average of 64kB memory for an embedded system, the middleware may consume up to a maximum of 16kB.

III. ARCHITECTURE

Existing middleware systems do not meet the requirements for the embedded systems domain as discussed in Section II. For example, platform and protocol support is often limited to a set of known standards and cannot be easily extended. Furthermore, the functionality of existing middleware solutions is often very elaborate, having a dramatic impact on the size of the software. This motivated us to develop a custom middleware layer, i.e., ActiveLink. This section describes the most important features of the ActiveLink architecture, using aspects as defined in [4].

A. *Interaction style*

ActiveLink provides a request-response interaction style based on remote procedure call (RPC), see Figure 2. An ActiveLink broker provides means to register services, to query remote services and to invoke remote services. ActiveLink does not provide mechanisms for locating or matching of services with requests. As a result, applications must know and explicitly specify the location of the application they need to communicate with.

Figure 3 shows the top-level architecture of ActiveLink. ActiveLink is a symmetric solution, i.e., identical modules run on both systems. These modules contain the broker and proxy functionality. An arbitrary network of applications can be configured by setting up multiple simultaneous peer-to-peer connections between them.

A feature in which ActiveLink differs from other middleware approaches is that it offers a programmers interface for remote memory management. Services are supported for memory allocation and de-allocation and memory copying between the local and the remote address space. These services enable client applications to freely inspect computational results on the server side, with only minimal impact on the remote system.

B. *IDL interface definition languages*

To span different languages, some middleware approaches require interfaces to be specified in a language-neutral Interface Definition Language (IDL). Typically, an IDL provides means for specifying method names, method types, and parameter types. In some cases, mechanisms exist to group interfaces into larger units.

ActiveLink supports only an interface to the C language. Hence there was no need for a full-blown IDL. During initialization of ActiveLink, a server application registers a list of functions that can be called remotely by client applications. The server application only has to register the function names. No information has to be provided about the number of function parameters or their types. ActiveLink can thus be called *weakly typed* in the sense that it is the responsibility of the client application to call a remote function with the proper number and type of parameters.

C. *Proxy*

To hide distribution, middleware systems create local programs, called *proxies*, to represent remote (either client or server) services. A client-side proxy and server-side proxy communicate with each other by transmitting requests and responses. The task of a proxy is to manage this flow of messages between clients and servers.

In ActiveLink, a remote function is called by passing the function name and parameters to the broker. The broker in its turn communicates with the remote system, after which the function will be invoked. The broker is responsible for returning the function result. Furthermore, the broker notifies the client application if it tries to invoke a remote function that is not registered.

In ActiveLink the proxy functionality is integrated with the broker functionality in one module.

D. *Marshalling*

Marshalling is concerned with the encoding of a request or response into a form suitable for transmission

across an infrastructure and for the decoding at a remote system. Difficulties arise when parameters are passed by reference to an other address space.

Even though ActiveLink is weakly typed, it is capable of dealing properly with function parameters and return values. All basic C types are automatically marshalled for function requests and responses. Compound types, like structure and union types, are not supported by ActiveLink and must explicitly be transferred by the client application. Besides this, the client application is responsible for properly handling references to other memory spaces.

E. Binding

The establishing of a connection between a client and a server is called the 'binding' of these parties. We speak of static binding when an application (either client or server) is compiled with knowledge of requested service interfaces (e.g. provided by an IDL specification). In general, applications do not have built-in knowledge of services available elsewhere in the system. Then, applications have to find out about other services at run-time. This is called dynamic binding and allows for more independent development and run-time extension of systems. However, it does require more overhead in the form of brokering mechanisms that typically incur larger execution cost.

In ActiveLink, no knowledge is required of the remote functions during compilation. Binding is performed during run-time. To overcome the performance penalty incurred by dynamic binding, a mechanism if offered that distinguishes reference retrieval from reference usage.

Reference retrieval takes care of the dynamic mapping between a service, i.e., a function name, and its reference. Reference retrieval should be done once and can be done when performance is not critical, e.g., during initialization. Subsequently, the retrieved reference can be used very efficiently to call the remote function. The retrieved reference can be reused as many times as required, without any further performance loss because of reference retrieval.

IV. CONCLUSIONS

In the development and deployment of embedded software, communication between multiple devices is often required. Middleware is a software technology that facilitates the communication between distributed applications. In this paper we listed key requirements for middleware for the embedded systems domain. Because current mainstream middleware does not meet these requirements, ActiveLink was developed as a solution for generic application-level cross-platform communication. ActiveLink has a number of features that make it especially suited for embedded systems:

- a very small memory footprint,
- remote procedure calling, and remote memory access and management,
- easily portable across platforms due to minimal platform dependencies,
- easily extensible with new protocols.

CMG integrated ActiveLink in its *Embedded TestFrame* architecture. This is a generic solution for the development of automated test suites for embedded software. The responsibility of ActiveLink in this architecture is to facilitate the connectivity between the embedded system under test and the host that stores the test suite. The Embedded TestFrame architecture is successfully being applied in several projects of CMG and of its customers.

Future enhancements to ActiveLink include support for user-defined interaction styles (such as streaming and events), removing dependencies on multi-threading operating systems, and support for dealing with unreliable protocols. Because extensions incur the risk of violating the footprint requirements, current research is focusing on development of a tailorable architecture where only the required modules can be used, leaving out other parts.

REFERENCES

[1] B. Venemans, *"Redesign of a flexible cross-platform communication utility"*, thesis for the Master of Technological Design in Software Technology program, Stan Ackermans Institute, Eindhoven University of Technology, The Netherlands, 2001.

[2] Harro S. Jacobs, Peter H.N. de With, *"Embedded TestFrame, An Architecture for Automated Testing of Embedded Software"*, Proceedings of the first PROGRESS workshop on Embedded Systems, The Netherlands, October 2000.

[3] R.J. Norman, *"Middleware: CORBA and DCOM"*, SDSU IDS Department Working Paper, October 1997.

[4] F. Plasil, M. Stal, *"An Architectural view of Distributed Components in CORBA, Java RMI, and COM/DCOM"*, Software Concepts and Tools, Springer, 1998.