Journal of Zhejiang University SCIENCE
ISSN 1009-3095
http://www.zju.edu.cn/jzus
E-mail: jzus@zju.edu.cn

CCPA: Component-based communication protocol architecture for embedded systems*

DAI Hong-jun (戴鸿君)†, CHEN Tian-zhou (陈天洲)†‡, CHEN Chun (陈 纯)

(*School of Computer Science, Zhejiang University, Hangzhou 310027, China*)

†E-mail: dahogn@zju.edu.cn; tzchen@zju.edu.cn

Received Nov. 11, 2004; revision accepted Feb. 1, 2005

Abstract: For increased and various communication requirements of modern applications on embedded systems, general purpose protocol stacks and protocol models are not efficient because they are fixed to execute in the static mode. We present the Component-Based Communication Protocol Architecture (CCPA) to make communication dynamic and configurable. It can develop, test and store the customized components for flexible reuse. The protocols are implemented by component assembly and support by configurable environments. This leads to smaller memory, more flexibility, more reconfiguration ability, better concurrency, and multiple data channel support.

Key words: Component-Based Protocol, Component-Based Protocol Architecture (CCPA), Dynamic configuration, Embedded communication system

doi:10.1631/jzus.2005.AS0079 **Document code:** A **CLC number:** TP393.04

INTRODUCTION

Communication is one of the most popular uses for embedded systems such as router, switch, mobile, PDA. In recent years, there are many hardware and software improvements to enhance communication. New generation embedded devices, following ARM 9, ARM 10 or Intel® XScale™ Micro-architecture, feature 32-bit data bus, up to 400 MHz processor, 64 MB SDRAM, 32 MB boot ROM, 32 MB flash memory (Iordache and Tang, 2003). These hardware environments can support more complex calculations and larger process space than ever. On the other hand, communication systems are typically structured into several layers, where each layer realizes a fixed set of protocol functionalities, such as well-known architectures of OSI stack or Internet Protocol Suite (Postel, 1996). The protocol stack in embedded systems is also layered and more modern protocols have realized such as the wireless protocols in mobiles.

However, embedded systems are still source-limited. They are generally restricted by computational capability and memory space. General purposed protocol stacks are not always adequate for the increasing demands of modern embedded systems. In particular, the demands on dynamic configurations and external maintenances are not well supported by existing protocol stacks.

Applications in one kind of embedded device often require various communication services because of differently linked devices or network environments (Bilek and Ruzicka, 2003). There are two common ways to resolve the problems. First, the systems are configured by operators to install and start the updated services without the system halt, but this requires that the systems have to stop for a time. Second, more services are activated to suit the various environments, but this increases the system work-load and needs more memory and storage in the devices (Swaminathan and Chakrabarty, 2004).

So a feasible approach is to decompose the

‡Corresponding author

*Project (No. 2002AA1Z2306) supported by the Hi-Tech Research and Development Program (863) of China

monolithic implementation of the software communication system, to structure protocols with protocol modules which can be reused in different contexts and be adapted to the special needs of applications. The protocols are self configured or remote configured with the module's replacement or increase. The systems load the new modules automatically and need not halt, called hot swap. The communication keeps alive and the updated process requires only a few seconds. Furthermore, a remote server stores and maintains all of the modules and only running modules load into the systems, unused modules can be hung up temporarily and even be deleted once they have been replaced. This saves the limited memory and space in the embedded devices.

The protocol modules can be implemented by some kinds of components. A recent software system paradigm, called component-based system (CBS), provides a new dimension of reusability and rapid prototyping of software. CBS focuses on dividing the software into components so that it can be easily configured and plugged together to assemble flexible applications. For example, AVOCA is a CBS communication system (Hempstead *et al*., 1992), which is a further stage of the *x*-kernel (Hutchinson and Peterson, 1991).

CBS gives rise to two new problems. First as mentioned above, the storage of components. A component library and a management server must be set up to deal with it. There are some differences compared with common database. Second, the rapid development of the components. Usually the analysis of existing programs is helpful to the further development and sometimes the development process involves only rebuilding the source codes to different EOS.

In this paper, we present the Component-Based Communication Protocol Architecture (CCPA) to modularize the implementation of component-based protocols. CCPA is neither component programming model nor binary compression method. It is the architecture for the development, storage and utilization of particular components for different EOS. These components are units of component-based communication protocols and especially suitable for embedded systems. We can only use the necessary components in the embedded devices with dynamic configuration and hot swap.

Using CCPA, we analyze the existing source codes of the protocols, develop and test qualified components targeted to different EOS. We devise multiple light-weight protocol components customized for application requirements and device characteristics. These stable and certified components are stored into a library for further reuse. Only necessary components are loaded into the devices and kept partly active according to the current network and environment status. The active protocols can be remote controlled without the communication being terminated.

Some applications from CCPA have been practically used for high performance CDMA2000 routers and high capability 10 G routers. For these complex systems, the specialized optimization shows higher performance and stability.

COMPONENT-BASED PROTOCOLS

Component model

Component theories have been widely spread from enterprise applications to operating system, web browser and middleware, etc. (Rastofer and Bellosa, 2001). The component, which is the independent and replaceable part of a system, fulfills a clear function and works in the flexible context of a well-defined framework. Traditionally, it is the binary format which is well defined by its interface and its operation. Fig.1 shows the encapsulation of the components. The interface has a set of ports for external or internal operations involving detailed implementations of the component functions.

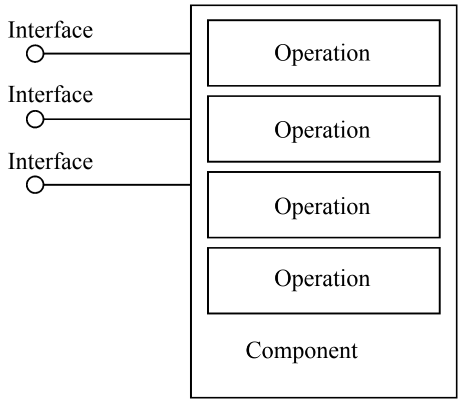

Fig.1 Encapsulation of the component

Component-based software usually involves certain code complexity and efficiency loss, and may be not essential for the common computers such as PC of x86 architecture, but it must be all-important for the embedded devices because of the low calculation speed, low network transmission speed and limited memory. So in embedded systems, modularized utilization appears recently. Especially because the underlying communication services keep active all the time, the components must be of high performance and critical small size. Therefore, common development and utilization methods are not universal for embedded systems, the development must be more precise for stability and the utilization must be more flexible.

Then the embedded communication system can be composed of efficient modules. The module is represented by the protocol component as the abstraction. Protocol components encapsulate the implementations of the protocol mechanisms, and provide the services outside through the interfaces. The communication protocols are manifested by unique interpretation of associations between components, and these components may request other ones through interfaces for assembly.

System support for components

The communication is regarded as the basic service, so the EOS must be updated to support the component-based protocol stacks from its kernel optimization. The model is shown in Fig.2. There is a boot component (BootCom) which can load other components with its inherent interface descriptions. A system service (ComS) co-exists when the EOS starts. It reads the configuration and loads the corresponding BootCom. BootCom assembles all of the other components according to the interface. The assembly processes watches with ComS until the component-based protocols go into action. Then the BootCom is hung up and inactive. ComS watches all the active components of this protocol and the configuration.

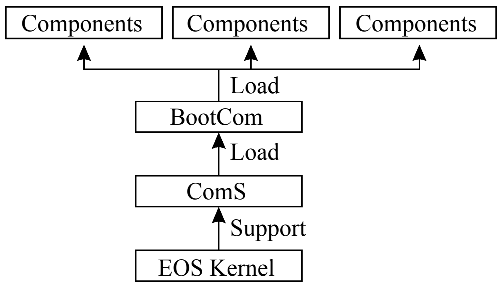

Fig.2 Component load model

For various communication devices and application requirements, the configurations of protocols may be distinct. Different protocol components are activated to serve the communication request and support run-time reconfiguration of the communication system. The states of the system are examined instantaneously. When it does not meet the request of the application, the suitable components will be loaded to replace what does not match. This enables the communication systems to adapt dynamically with application requirements such as switching from unreliable to reliable data delivery, communication system resources such as buffer space and CPU load, or network characteristics such as network congestion and routing.

The replacement of protocol component is transparent with the interfaces unchanged. This is one of the frequent mechanisms supporting the hot swap. The operation in components may be changed but the assembly state is changeless because of no interface change. So the new communication components are dynamically loaded by ComS with the mapping to the new component interface address, BootCom is still not active. Another case, If the interface needs to be changed, BootCom is updated first when it is inactive. Then ComS loads the new BootCom, other components are updated by BootCom.

Protocol decomposition

We may decompose the protocol from the protocol hierarchy. For example, the connection oriented transport system of the Internet Protocol Suite consists of three components: TCP, IP and Ethernet. Otherwise, we may base on protocol functions to break down the protocol. The protocol functions serve as basic CCPA components too. Taking TCP as an example, the sender part located between the application and network interface is totally composed of five components, which are connection management, data emit, data reemit, flow control and congestion control (Schmidt *et al.*, 1993).

Protocols may be implemented with different mechanisms too, such as the former example's TCP, flow control which could be realized by a window-based or rate-based mechanism. Table 1 lists an example for protocol functions and corresponding mechanisms. Different mechanisms of one protocol function are implemented by the corresponding components while the interface remains the same.

Table 1 Protocol functions and corresponding mechanisms

Protocol function	Mechanism
Flow control	Stop-and-wait
	Window-based
	Rate-based
Corruption control	Checksum
	Parity
Connection management	Implicit
	2-way-handshake
	3-way-handshake
	No connection (datagram)
Acknowledgement	Cumulative
	Selective

One of the key goals of CCPA is that we can implement applications with device-customized modules for the same communication. New enhancive modules can be added in by writing additional modules and including them in the existing component suite. For example, to the components of Internet Control Message Protocol (ICMP), we can give more suitable modules in different environments. When the network's bandwidth is low, the proper module to use is whose numerical value of Time to Live (TTL) fits the network status. Timestamp Reply is set to be forbidden so as to decrease the overload.

We can modify original properties of the components and add customized communication services into the modules too. For example, the Denial of Service (DoS) attack is often caused by the mechanism of sending large numbers of useless packages. It will be helpful to design a module that resolves the problem by checking the source/destination address and the type of the package, including the package frequency. Then the component of ICMP will have the ability to filter and check the package. With all these modules, customized communication services can be constructed and provide execution guarantees targeted to the specific requirements of the applications on different devices.

CCPA DEPLOYMENT

CCPA includes a set of correlated software systems, such as Component Development Platform (CDP), Component Library (CL), Component Assembly Platform (CAP), and Operating System Support Environment (OSSE). CAP is optional software based on different device configuration demands.

Component Description Language (CDL) is defined with XML format to describe the communication services, the component-based protocols and the characters of the components. Each component and each protocol has its own description. If the components are generated by CDP, the descriptions are written by the developers or test units as a part of test reports. The descriptions join these software systems together, encapsulate the component operations and expose the component interfaces accurately. They are also the mapping index of CL, the assembly rules of CAP and the execute parameters of OSSE.

A general scenario of the architecture is given in Fig.3. CDP is an IDE which is used to develop the components suitable for EOS. The qualified components through tests are stored into CL. OSSE executes on EOS or joins into EOS kernel to supports the component-based protocols and the protocol stacks. According to application requirements or network environment, it can load or unload the components while the communication service is kept active. CAP is used to control OSSE remotely, so it can get the current configuration of OSSE. It must keep the synchronization with CL to know the usable components.

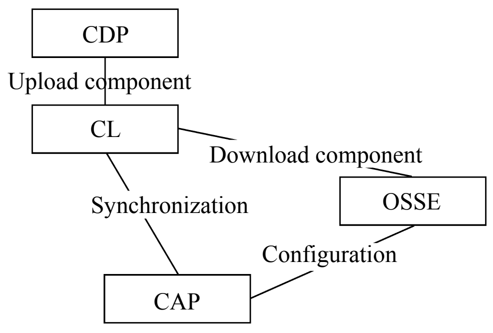

Fig.3 CCPA basic deployment

Component development

How to rapidly develop or rewrite protocols and protocol stacks to support component-based system, how to generate the qualified components, are both key problems. This is the source generating the reusable components. CDP is an IDE used to aid the development process. Besides the basic functions such as Visual Studio and KDevelop, it has own conveniences for CCPA. Code analysis and cross-platform tool-chains are the significant characters.

During the implementation of the protocol stacks, often it is the transplant of stable protocol stacks to Embedded OS (EOS), such as part transplant TCP/IP implementation from Linux to fit the Embedded Linux. Most source codes of common OS are not customized for EOS and take little care of the restricted computational capability and memory space, which may cause problems if the transplant only focuses on executable and receptive program as the performance and efficiency becomes secondary. Although the loss of performance is a matter of fact, this may be unpredictable because of lack of criteria. So we must guide and help the analysis of existing protocol stacks and source codes in CDP. Especially for wireless protocols, the repetitive transplant among different EOS in different devices gets more troublesome. Indeed, most of the common communication protocols have their robust C version source codes. Code analysis is important for efficiency as the rewriting work does not build up from nothing.

CDP is an IDE used on the common PC whose OS is Windows or Linux. The source codes are edited and managed on a PC, but the generated components are run in EOS. Cross-platform compilers must be used to compile the source codes and must use certain cross-platform tool-chains such as compilers and debugging tools, which are customized to target EOS. So the universal link interface is necessary to connect all kinds of them; this makes the development process fully oriented to the programs without considering tool-chains.

The quality and reusability of the binary components must be checked. In CDP, it can be tested and examined from the source code level. If there are emulation environments of the target EOS and devices, further tests such as those for component independence, assembly stability, and component running memory, can be conducted to get accurate component descriptions and evaluations. Only qualified components are uploaded to CL.

CL is not merely a binary database for component storage. It does much quality assurance, exchange and component transfer work. Before being stored into the database, each component must pass the ultimate examination which focuses on the integrality and uniform external interface. For the components which do not upload from CDP, this quality assurance is vital. The components must pass the examinations before they are stored into the libraries. The quality of components is the key assurance of CCPA. This makes active services on embedded systems stable and robust.

Configurations and utilizations

Furthermore, CL is the center of the components and the descriptions of the component-based protocols. It may be distributed and there must be a built-in mechanism to keep synchronization between libraries. It passes the component and protocol descriptions to CAP for manual configuration and assembly. It is also the only sources of the components for OSSE.

OSSE executes on EOS or joins into EOS kernel. The chief work is to support the component-based protocols and protocol stacks. At the same time, it provides the protocol runtime watch and dynamic configuration through the ComS which was mentioned above. It can self-adapt to the application requirements and network environment, or can be manually controlled remotely by CAP. It loads or unloads the components while keeping the communication services active. CAP is an optional IDE. Users can do initialization and configuration work manually here, according to the CL contents and the requirements.

Fig.4 shows the lifecycle of the correlative work with CAP. CL is a server that runs all the time. CAP connects to the CL to get the protocol and component descriptions and also connects the OSSE to get the active components information. The configuration

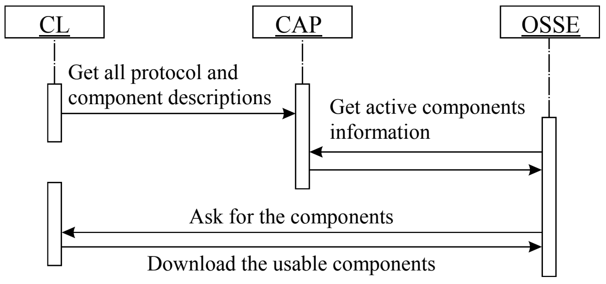
CL
CAP
OSSE
Get all protocol and component descriptions
Get active components information
Ask for the components
Download the usable components

Fig.4 The lifecycle of the correlative work with CAP

processes on the CAP and the result returns to OSSE. Then OSSE connects to CL for the components. CL validates the requests and the components stored in CL which have passed the quality verification can be downloaded to OSSE. OSSE runs the updated components as discussed in the last chapter. If there is no CAP, the correlated model is simpler.

In contrast, Avoca (Hempstead *et al*., 1992) is a network architecture and domain model that supports the development of encapsulated, reusable, and efficient communications protocols. The runtime environment for Avoca is provided by the x-kernel which is an operating system kernel designed to run network protocols. The domain model for Avoca focuses on the identification of realms of protocols for remote procedure calls, remote invocation methods, and network file systems. Avoca protocols are symmetric components that can be composed in virtually arbitrary orders (Batory and O'Malley, 1992). According to the introductions above, it can be concluded that the implementations of CL and CAP, and the correlations among them are the main character of CCPA, which also includes CDP to ensure the component quality. These designs adapt to the limited computational capability and memory space of embedded systems. They also have more flexibility, such as dynamic configuration and hot swap.

IMPLEMENTATION AND RESULT

There are some IDE, such as Visual C++ "ATL COM appwizard" and Borland Java Builder "Object Gallery: JavaBeans, EJB, Cobra module wizard", which can guide the component development. There is CL which can store many types of components such as Jade Bird Component Library System (Mei *et al*., 2000) too. But only CCPA is fully customized for communication protocols in embedded systems. We built a set of complete implementation based on CCPA and made some tests on an Intel® PXA 255 hardware platform with a built-in 10~100 M self-adaptive network card.

For example, FTP is an application layer protocol. A simple implementation of FTP client is about 1100 lines source codes of C language. Through analysis tools of CDP, it is easy to find that the codes are generally composed of three main parts: transfer part, control part and upload part, so we can divide this FTP implementation into three modules according to its own function, named C_Transfer, C_Control, and C_Upload. These components can make up the FTP service.

We built two sets of components suitable for different network bandwidth. First, the optimized work exerts good influence on the source codes for 10 M network bandwidth. Three components were built: C_Control, C_Transfer_10, and C_Upload_10. Second, the codes are optimized for 100 M network bandwidth. We got two new components: C_Transfer_100 and C_Upload_100 (C_Control is nothing different). All these components pass complex and strict tests before they are stored into CL.

Then OSSE in embedded devices download the suitable components from CL to assemble the component-based FTP client. The configuration comes from CAP. Then we use this FTP client to upload files, using Average Transfer Speed (ATS) as test data statistics. ATS is defined by recording transfer speed every 15 s, and taking the average after getting at least 10 values. The following steps yield the value in Table 2.

First, the cable is 10 M available and we use the original FTP client implementation which is directly compiled with the original C source codes. We can get the data of Row 1. Then we use the components suitable for 10 M network bandwidth, the result is Row 2. If the cable is changed to be 100 M available, the result is Row 3 and the components update automatically by OSSE self-adaptive mechanism.

Second, we test the dynamic configuration and the cable is 10 M available. If the unique C_Upload_10 is unloaded while it keeps actively uploading files, the result is shown in Row 4. After the C_Upload_10 is loaded again, the result is shown in Row 5. After the transfer process becomes stable, if the C_Transfer is unloaded, the result is shown in Row 6.

Third, we add multiple C_Upload_10 to execute at the same time. Test data varies from Row 7 to Row 9.

Comparison of Row 1 data with Row 2 data shows that there is certain performance loss after the FTP service is component-based, but the loss is no more than 8%, so it is acceptable. Comparison of Row 2 with Row 3 shows that the active components vary

Table 2 Simple component-based FTP service test data (The scale of ATS (Average Transfer Speed) is 5 kB)

		Active component			*ATS* (kB/s)
		C_Transfer	C_Control	C_Upload	
1	Use original application	No	No	No	255
2	Use 10 M cable	C_Transfer_10	C_Control	C_Upload_10	240
3	Use 100 M cable	C_Transfer_100	C_Control	C_Upload_100	770
4	C_Upload unload	C_Transfer_10	C_Control	No	0
5	C_Upload load again	C_Transfer_10	C_Control	C_Upload_10	240
6	C_Transfer unload	No	C_Control	C_Upload_10	240
7	Use 2 C_Upload	C_Transfer_10	C_Control	2 C_Upload_10	325
8	Use 3 C_Upload	C_Transfer_10	C_Control	3 C_Upload_10	350
9	Use 4 C_Upload	C_Transfer_10	C_Control	4 C_Upload_10	360

automatically accompanying the cable change, the transfer speed increases so much mainly because of the better network environment.

Row 4 shows that if the component for corresponding function is unloaded, the activity pauses at once, and Row 5 shows that if the component is loaded again, the activity continues and is not influenced. After the transfer activity becomes stable, the unused components can be unloaded with no influence as shown in Row 6.

From Row 7, it is obvious that the performance has increased by about 40% if we add a functional component. But from Row 8 to Row 9, it can be concluded that the increase of performance becomes much slower even if more active components are added into the system. The status may be much better if we use multiple CPUs.

An analysis chart of test data is shown in Fig.5. Line 2 is Row 1 and Line 1 is Row 3. We can find obviously that:

(1) Value of Row 2 is a little smaller than that Line 2, because of the loss of the component-based FTP service.

(2) There is the remarkable increase from Row 2 to Row 7, because of the addition of an upload component

(3) There is still increase from Row 7 to Row 9, but the trend is moderate, because of the limit of software improvement.

(4) Line 1 is much higher than the others, because of the huge change in network status. If the update components cannot load dynamically, it is so much waste.

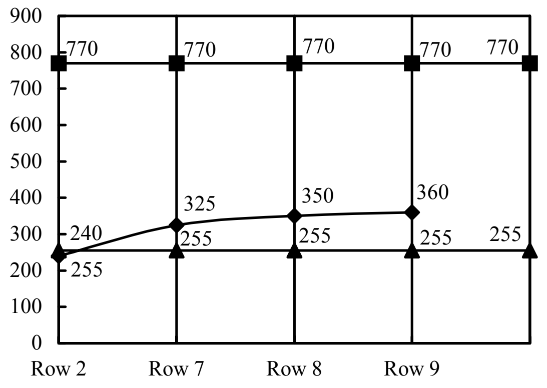

Fig.5 Data analysis of an Ftp client

CONCLUSIONS AND DISCUSSIONS

For the sake of the efficiency and performance of communication protocols, the component-based communication architecture (CCPA) is constructed to realize and improve the component-based communication service. It enables the dynamic configuration of customized protocols through a proper set of reusable protocol components assembly. The replacement of protocol components doesn't need to reboot the device. To produce, test, store and use the components, CCPA is deployed with a set of software including component development platform, component library and component assembly platform etc. The flexibility and reusability brought by the architecture are obvious and recommendable.

Some EOS is component-based such as TinyOS (Hempstead *et al.*, 2004) and merges the protocol stack into the EOS kernel, so that one component may be multiple reused by the component-based protocols in the same embedded devices and the performance may be much higher in the multiple CPU systems such as SMP devices.

References

Batory, D., O'Malley, S., 1992. The design and implementation of hierarchical software systems with reusable components. *ACM Trans. on Software Engr. and Methodology*, **1**(4):355-398.

Bilek, J., Ruzicka, I.P., 2003. Evolutionary trends of embedded systems. *IEEE International Conference on Industrial Technology*, **2**:901-905.

Hempstead, M., Welsh, M., Brooks, D., 2004. TinyBench: The Case for a Standardized Benchmark Suite for TinyOS Based Wireless Sensor Network Devices. 29th Annual IEEE International Conference on Local Computer Networks, p.585-586.

Hutchinson, N., Peterson, L., 1991. The *x*-kernel: Architecture for implementing network protocols. *IEEE Trans. on Software Engineering*, **17**(1):64-76.

Iordache, C., Tang, P.T.P., 2003. An Overview of Floating-point Support and Math Library on the Intel/spl reg/ XScale/spl trade/architecture. 16th IEEE Symposium on Computer Arithmetic, p.122-128.

Mei, H., Xie, T., Yuan, W.H., Yang, F.Q., 2000. Component metrics in Jade Bird Component Library System. *Ruan Jian Xue Bao*, **11**(5):634-641 (in Chinese).

Postel, J., 1996. Internet Official Protocol Standards, RFC 1720. Network Working Group.

Rastofer, U., Bellosa, F., 2001. Component-based software engineering for distributed embedded real-time systems. *Software, IEE Proceedings*, **148**(3):99-103.

Schmidt, D.C., Box, D.F., Suda, T., 1993. Adaptive: A dynamically assembled protocol transformation, integration, and evaluation environment. *Concurrency: Practice and Experience*, **5**(4):269-286.

Swaminathan, V., Chakrabarty, K., 2004. Network flow techniques for dynamic voltage scaling in hard real-time systems. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, **23**(10):1385-1398.