LOOSELY COUPLED TOOL INTEGRATION ARCHITECTURE FOR EMBEDDED LINUX

Seung Woo Son, Hyung–Taek Lim, Chae–Deok Lim, Heung–Nam Kim
Embedded Software Research Team
ETRI–Computer & Software Research Laboratory
161 Gajeong–dong, Yuseong–gu, Daejeon, 305–350, KOREA
{swson,htlim,cdlim,hnkim}@etri.re.kr

Abstract

When developing applications under embedded Linux, there is a range of software tools to assist the development process. However, they are so tightly integrated that it is not only difficult to integrate into IDE but also hard to add an additional tool. They are communicating with each other through their own protocols so that there is much communication overhead between host and target machine. In this paper, we propose loosely coupled tool integration architecture based on lightweight RPC mechanism. The proposed architecture is mainly composed of two agents running on host and target side respectively. To integrate various tools in a seamless way, we suggest well–defined host–to–target protocol and inter–tool protocol. Host agent mediates request from several host tools and interacts with the target agent. The host agent set out a set of common APIs needed for host–resident tool integration. The target agent, the counterpart of the host agent, is a kind of debug demon which servers request from the host agent. As designed for embedded software development, the size of target agent is about 108KB and consumes less than 5% of CPU time. With the open tool architecture, we easily integrate various host tools using open APIs.

1 Introduction

Embedded Linux is getting popular in embedded system market because of its flexibility of use and various target support without any royalty [6]. When developing embedded applications under embedded Linux, there are many tools that support the development process. Industry's best–known free software development tool is GNU toolkit. However, the components in GNU toolkit were originally designed for developing desktop applications in a Unix–like environment [2]. These inherent limitations made developers hard to choose a Linux as their embedded software solutions.

In traditional embedded software development environment like Tornado [11], Spectra [8] and Esto [3], there are already their own IDEs which make it easy to develop embedded applications(IDEs are dominating tool). In Linux side, there exist two ways of approach in embedded Linux toolkits. One is simple approach that relies heavily on the GNU tools. The other is complex approach that provides GUI tools. However, the user interface is a kind of GUI/command line integration, so that it is difficult for a new Linux developer to become accustomed to the environment [1].

In addition, these tools are tightly coupled, so that it is difficult to add third–party tools [7]. Tightly coupled means that each tool in development toolkit communicates with each other with its own communication mechanism. Even a rapid change in Linux kernel and its development tools make it difficult for commercial tool vendors to adapt a powerful IDE based on open architecture.

In this paper, we propose loosely coupled tool integration architecture for embedded Linux. The proposed architecture is the extension of our previous tool architecture for embedded Linux with same tool architecture. We adapted many features from traditional architecture and made some modification on it.

This paper is organized as follows; Section 2 discusses the tool integration architecture. The design and implementation of the proposed architecture is described in section 3. Section 3.1 and section 3.2 describe the internal structures of our architecture. The evaluation of the proposed architecture is described in section 4. Finally, conclusion and future

work are discussed in section 5.

2 Tool Integration Architecture

Even in development of embedded software, IDE improves productivity and quality of code. To provide embedded programmers with flexibility in choosing tools and targets with a IDE, underlying tool architecture is important. In terms of tool integration architecture, there are three options [7]:

- tightly integrated/closed system,
- loosely integrated/open system,
- distributed object approach.

In tightly integrated system, if a tool wishes to talk to another tool then a one–to–one communication link is established between the two tools. This approach meets considerable user dissatisfaction because it is difficult for users to add an additional tool [7]. Many toolkit for embedded Linux, like Red Hat's ELDS [1], choose the closed architecture.

In the loosely integrated system, the integration involves tools communicating via a 'controller' which is central to the IDE. This approach accommodates both first and second level integration, but it requires the definition of a communication mechanism between each tool and the controller [7].

In traditional embedded software development, there are already many IDEs which facilitate the development process. Overall ease of using the development environment is an important factor in choosing the RTOS and its toolkit. For example, VxWorks [12] has earned its popularity in embedded development tool market because of its powerful IDE, Tornado [11]. Many commercial IDEs adapt the open architecture for better productivity and easy of use [13]. In architectural view of point, the distributed object approach may be the most suitable option. However, it is difficult to adapt to a remote development environment because using COM or CORBA as communication mechanism may be too heavy for the target machine which has usually limited computing resources.

A growing number of embedded Linux tool vendors are now offering toolkits with IDE. However, they are mainly focusing on simplifying the task of embedding Linux.

3 Design and Implementation

This section explains the design and implementation of our tool integration architecture. The architecture is originated from the architecture for RTOS in which kernel and applications tasks (or threads) are running on a single memory space like VxWorks [12], VRTX [8] and Qplus–T [4, 5, 3, 9]. We modify our previous architecture, so that new architecture is appropriate for embedded Linux.

The central ideas of the proposed architecture are as follows:

- Minimal overhead on the target.
- Centralized communications with the target.
- Open architecture to make it easy to add third-party tools.

The following two agents are important, because they mediate all contact between the host–resident tools and the target:

- The TA(Target Agent) is the target–resident component that handles request from host tools.
- The HA(Host Agent) is residing on a host machine and manages communication between various host tools and a TA.

Figure 1 illustrates the role of these two agents in our architecture. With these two agents architecture, we can support embedded programmers for embedded Linux with same tool architecture. Under the Linux development environment, tools in Figure 1 include gcc, gdb [10], remote shell, some target monitoring or profiling tools, etc.

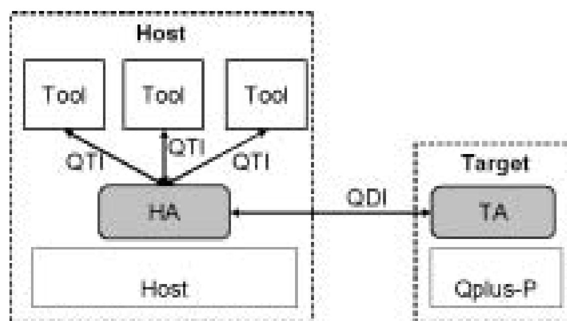

FIGURE 1: *HA, Tools and TA*

3.1 TA and host–target interface

Host–target interface has almost same protocol as the previous interface, QDI(Qplus Debug Interface) [9]. The TA carries out requests transmitted from the HA, and replies with the results. The TA contains a compact implementation of UDP/IP, which supports an RPC messaging protocol called QDI.

The QDI protocol is a minimum set of the services necessary to respond to requests from the host tools. These protocol requests include memory read/write operations, breakpoint/event notification services, and process control. The QDI protocol uses the Sun Microsystems specification for External Data Representation (XDR) for data transfer. Since the TA is also a process running on embedded Linux, the TA can execute in user mode. This limits debugging aspect of an embedded application to an application–level debugging.

Figure 2 illustrates the internal structure of TA.

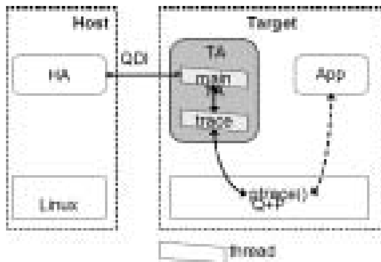

FIGURE 2: *Internal structure of TA*

The trace thread is in charge of processing ptrace–dependent services for a process. The main thread initializes TA when TA starts up and repeats receiving QDI request from HA, dispatching a QDI function, handling ptrace–independent services, and returning results to HA.

If an application program is to be traced in target, TA's main thread creates a trace thread, which in turn forks an application process. If more than one application program is to be traced, as many trace threads are needed.

Upon receipt of a ptrace–dependent QDI request, the main thread forwards it to the trace thread. Then, the trace thread processes the request and returns results to the main thread, which in turn returns results to HA. While the trace thread is waiting for stop signal from of the application process, the main thread can handle other QDI requests.

The main and trace thread use global variables to share data, which results in almost no memory copy, and use semaphores to synchronize them.

3.2 HA and Inter–tool Interface

The HA runs on host systems. All tools access a target through HA, whose responsibility is to satisfy each tool's requests. The HA manages all of the details of communicating with the target, so that tools are not concerned themselves with host–target transport details. The host tools use the QTI protocol [4, 5] to communicate with HA.

The HA includes following modules to support inter–tool interface [4, 5].

- Session management
- Information support
- Event management
- Debugging support

Since the RTOSes like VxWorks [12], VRTX [8] and Qplus–T [5] are running on single space memory system, such management modules for object module, symbol table and target memory are required. However, since Linux has file system, the object module loading is not required. HA's role in our architecture is a communication mediator between host–resident tools and TA.

Figure 3 illustrates the internal structure of HA.

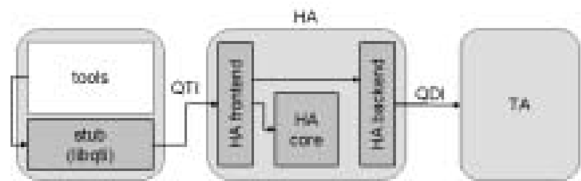

FIGURE 3: *Internal structure of HA*

The HA is mainly composed of four components; stub, front–end, core, and back–end. The stub is a static library(libqti) and each tool communicating with HA must be statically linked with it. If a tool call a stub function, the stub function calls the front–end function. The front–end function can use make use of HA's core function. If the request from a tool needs communication with TA, that front–end function calls a back–end function which sends QDI protocol to TA.

4 Evaluation

The low overhead on the target is important because the target system is usually a low–end device. Our approach adapt lightweight RPC for back–end module, so that it put low overhead on the target. The binary size of TA is just 108KB and consumes about 5% of total CPU time. These small size and low overhead target daemon is appropriate for low–end devices, like PDA or WebPAD.

Our architecture encourages users and third parties to customize our environment and to add features to it. The proposed architecture is structured to make adding tools easy.

Following is the procedure to add a new QTI and QDI under our architecture.

1. define a new QTI and QDI protocol to add,

2. define a data structure that a new protocol will use,
3. define function,

A new QDI and QTI is defined in the corresponding header file as below.

```
#define QTI_FILE_PUT 200
#define QDI_FILE_PUT 200
```

If there exists parameters that the stub and front–end function will use, users should define the parameters in structure data type. To add a new protocol into HA, users should define four functions; stub function, front–end function, back–end function, and XDR function. As depicted in Figure 4, the stub function calls front–end function. The front–end function is divided into QTI service function (starting with *qtisvc*) and QDI service function(starting with *tgt*). The front–end function calls back–end function whose name starts with *rpcCore*.

Comparing the execution time of steping over memcpy() statement with gdb/gdbserver, our approach is 4 msec slower than that of gdb/gdbserver. Stepping over assignment statement is less than 1 msec slower. We believe that several msec of delay is tolerant to the users considering the easy of adding a new tool.

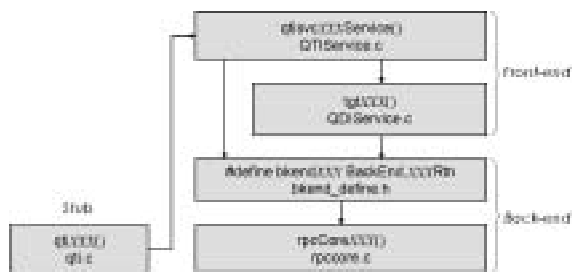

FIGURE 4: *The relation among stub, front-end and back end*

5 Conclusion and Future Work

In this paper, we propose a loosely coupled tool integration architecture under embedded linux. The open system has advantages over the closed system in that it is appropriate for extending IDE and adding a new tool easily. Since our architecture is the base of our IDE for embedded software development, lower overhead on the target is important. We attained the lower overhead on the target via UDP with XDR. Though it is a bit slower than just using gdb/gdbserver as a debugging environment, our architecture is scalable than gdb/gdbserver.

Linux provides many advantages of an open source distribution model. But along with many advantages of Linux come the usual drawbacks of an open source project. The rapid changes in Linux itself make it difficult for tool vendors to maintain their development tools within their own IDEs. We will extend our architecture tolerant to rapid change in development tools in Linux environment.

References

[1] Jerry Epplin. A developer's review of red hat's embedded linux developer suite. *http://linuxdevices.com/*, Nov 2001.

[2] Bill Gatliff. Embedding with GNU: GNU debugger. *Embedded Systems Programming*, pages 80–94, Sep 1999.

[3] K.Y. Lee and et al. A design and implementation of a remote debugging environment for embedded internet software. *ACM SIGPLAN Workshop on LCTES*, Jun 2000.

[4] C. D. Lim and et al. A tool broker in remote development environment for embedded applications. *IASTED AI*, pages 757–761, Feb 2000.

[5] C. D. Lim and et al. Middleware for platform independent toolset to develop real-time embedded applications. *RTCSA*, pages 49–53, Mar 2002.

[6] John Lombaro. *Embedded Linux.* New Riders, 2002.

[7] L. P. Maguire, T. M. McGinnity, and L. J. McDaid. Issues in the development of an integrated environment for embedded system design, Part B: design and implementation. *Microprocessors and Microsystems*, pages 199–206, 1999.

[8] Microtec. *VRTXsa Real-Time Kernel: Programmer's Guide and Reference.* Microtec, 1997.

[9] S. W. Son and et al. Debugging protocol for remote cross development environment. *RTCSA*, pages 394–398, Dec 2000.

[10] Richard Stallman. *Debugging with GDB.* Cygnus, 2002.

[11] WindRiver Systems. *Tornado User's Guide 2.0.* WindRiver Systems, Inc., 1999.

[12] WindRiver Systems. *VxWorks Programmer's Guide 5.4.* WindRiver Systems, Inc., 1999.

[13] Peter Varhol. Integrated software tools improve productivity and code quality. *Electronic Design*, pages 62–70, October 1999.