中華民國八十六年全國計算機會議

PROTOOL – Sun RPC 程式設計的自動化工具

PROTOOL – An Automated Development Tool for Sun RPC Programming

焦信達，蔡銘堂，袁賢銘
Hsin-Ta Chiao, Ming-Tone Tsai, Shyan-Ming Yuan

交通大學資訊科學系
Department of Computer and Information Science
National Chiao Tung University
1001 Ta Hsueh Rd., Hsinchu, Taiwan 30050, ROC
E-mail: {gis84532, gis82558, smyuan}@cis.nctu.edu.tw

摘要

Sun RPC 是目前被使用得最廣的一種 RPC 機制。但它仍有以下幾個缺點：首先，它對程式設計師來說，並不是完全透明的。而且，它產生出來的 iterative server stub code 的效率也不好。針對這兩個問題，我們設計了 PROTOOL，它可以讓完全不懂 Sun RPC 語法的人，也能利用 Sun RPC 機制設計分散式程式；另外，PROTOOL 能產生效率更好的 concurrent server stub code，並自動幫程式設計師處理 concurrent server 上 concurrency control 的問題。根據我們的測量，由 PROTOOL 調校的 concurrent server 程式，與由程式設計師自己動手修改的 concurrent server 程式，在效率上極爲相近。

關鍵字：分散式系統，遠端程序呼叫，並行性控制。

Abstract

Sun RPC is the most popular RPC implementation. However, it has some shortcomings. First, it is still not fully transparent to programmers. In addition, it generates inefficient iterative server stub codes only. To overcome these problems, we designed a development tool, called "PROTOOL". It can help programmers that know nothing about the syntax of Sun RPC to write distributed programs based on Sun RPC infrastructure. Moreover, it can generate stub codes for concurrent servers and handles concurrency control problem introduced by concurrent servers automatically. According to our measurements, the performance of concurrent server programs enhanced by PROTOOL are all most the same with the performance of handcrafted ones.

Keywords: distributed systems, remote procedure call (RPC), concurrency control.

1. Introduction

Distributed systems offer many fascinating features such like better cost/performance ratio, higher reliability. However, their software structure is usually very complex. Programmers have to deal with issues of communication, concurrency control, heterogeneousity, etc. Remote procedure call (RPC) [1] is one of the efforts that try to solve this problem. It models communications between clients and servers as conventional procedure calls. The details of communication are handled by client stubs and server stubs, which were generated by IDL compilers according to interface descriptions. To write distributed programs using RPC, programmers simply undertake the fellow steps:

1. Build a conventional program.
2. Split its source codes into two parts, namely, client part and server part.
3. Write an interface specification.
4. Generate stubs using IDL compilers.
5. Compile and link the client and server separately with its corresponding stubs.

This is really much simpler than building it from scratch but still not fully transparent to programmers. First, they have to write an interface specification by themselves. Second, some IDL compilers generate stubs for concurrent servers, within which the number of remote procedures that can be executed at one time is not restricted. As a result, race condition [5] may be introduced by the concurrent activities within it. To avoid this problem, programmers must write some codes to synchronize accesses to shared resources. Third, it is sometimes necessary to initialize these resources and requires programmers to modify server stubs manually. Thus, developing a distributed program is still like a race of obstacles.

Shani and Gold [9] proposed a development tool called "MakeDCE" for OSF/DCE RPC to generate interface specifications and binding codes

automatically. However, it does not emphasize the server synchronization problem even though the IDL compiler on DCE generates concurrent server stubs. It does not handle the server initialization problem, either.

In this paper, we focus on the Sun RPC[7]. It is the most popular RPC implementation. However, it introduces new problems. First, because of its version control and the restriction of parameter passing, stubs of it are comprised of two parts: a communication stub and some interface procedures. An IDL compiler, called "rpcgen," generates communication stubs from an interface specification. To supply interface procedures, programmers should do more than they have to do on other platforms. Second, the rpcgen can generate only iterative servers; that is, at most one remote procedure can be executed at one time. Although it avoids the race condition discussed above, it will have negative impact on performance, especially for servers that perform I/O frequently.

To solve these problems, we design a GUI-based automated development tool called "PROTOOL." Programmers can use it to build a distributed program that contains multiple clients and one server. Its inputs are configuration files. Each configuration file corresponds to a client or to a server. Writing a configuration file is so simple that programmers need no knowledge about Sun RPC syntax. The PROTOOL generates an interface specification and interface procedures automatically with the aid of configuration files. Moreover, it generates not only iterative server stubs, but also concurrent server stubs. To prevent race condition happened in concurrent server, it automatically generates codes for synchronizing the accesses to shared resources. It also handles server initialization in a novel way. In summary, programmers only have to know how to separate their programs, and the remains are left to PROTOOL. This will reduce burdens of RPC programmers further.

2. System Overview

In this section we describe how the PROTOOL works. The details of concurrent server will be discussed in Section 3.

Figure 1 depicts the components that constitute the PROTOOL. We create all these components except the rpcgen. How the PROTOOL works is described below:

First, users designate the configuration files, the network protocol, and the server type (iterative or concurrent server) through PROTOOL's GUI.

A client configuration file contains a list of source files that constitute the client. Similarly, a server configuration file also contains a list of source files that constitute the server. By default, all server procedures that are triggered directly by client parts will be exported. Programmers can also control the export list manually through server's configuration file. PROTOOL will generate interface description for these exported procedures automatically. In addition, a server configuration file may contain group membership of library procedures (discussed in section 3.2), and a list of procedures that initialize the server (discussed in section 3.3).

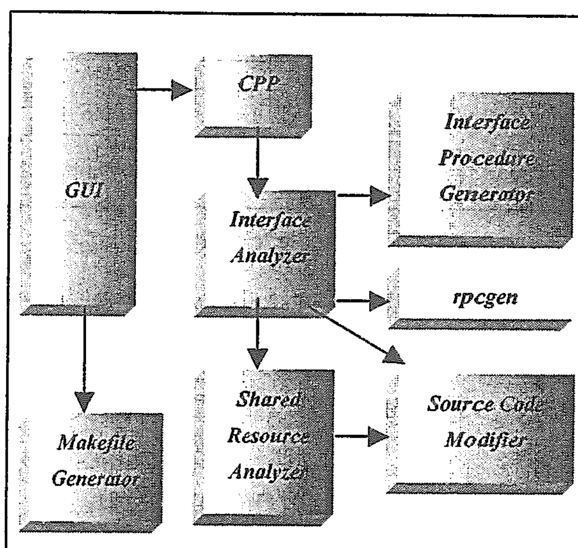

Figure 1: The Components of PROTOOL

After PROTOOL gathered these inputs, it invokes a compiler preprocessor (CPP) that deals with CPP directives, such as #include, #if, #ifdef, and so on, within all source files. At the meanwhile, a makefile generator creates makefiles for each client and server.

After the CPP processes all source files, an interface analyzer will be start up. It first scans all source files of clients to build a list of procedures activated by clients. Next, it scans all source files of the server to build another list of procedures defined on the server. By comparing these two lists, it generates a list of server procedures triggered directly by clients. Then, it modifies the list by referring to the server configuration file. This list is called interface. In the last step, this interface is translated into the format of Sun RPC's interface specification that will be used by the interface procedure generator and rpcgen. After interface analyzer finished its job, these two components will generate all interface procedures and communication stubs.

If users wish to have a concurrent server, a shared resource analyzer will be executed. Otherwise, this step will be skipped. It parses all source files of the server to understand how these exported server procedures access shared resources. After the shared resource analyzer ends, a source code modifier will be invoked. It modifies default iterative server stubs and makes them become concurrent server stubs. Besides,

it also adds synchronization codes to server's source files automatically.

After all things done, users can use these makefiles to produce executable programs. Figure 2 shows an example of the input and output files of the PROTOOL.

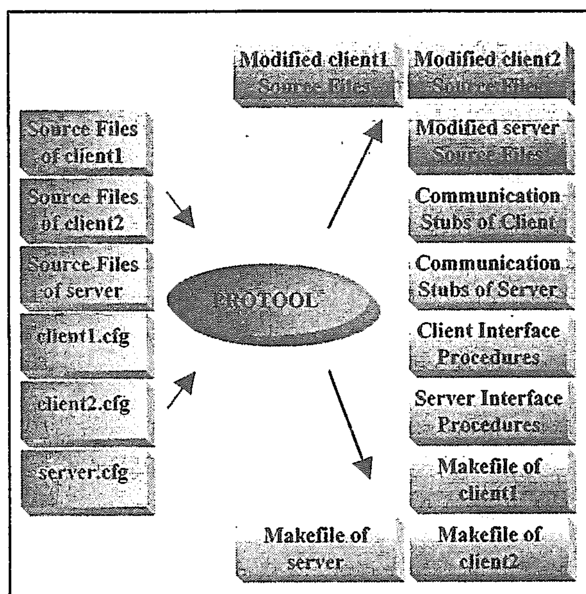

Figure 2: The Input and Output Files of PROTOOL

3. Issues of Concurrent Servers

This section describes the details of the shared resource analyzer and the source code modifier. First, we introduce how the source code modifier changes iterative server stubs to concurrent server stubs. Later, we describe how to avoid race condition and deadlock problem within a concurrent server. Finally, we discuss how to initialize servers.

3.1 Server Stubs for Concurrent Servers

Let us investigate how the source code modifier takes advantage of LWP library [8]–an implementation of thread packages on SunOS–to make RPC servers concurrent.

We developed a new request dispatcher, i.e., svc_**run**( ), to replace the old one generated by rpcgen. This function is called by server stub after server registers itself to DNS server and endpoint mapper. It has a dispatching loop to dispatch any valid remote procedure calls from clients.

Our new svc_**run**( ) is shown in Figure 3. Before the while loop begins, it allocates a private stack for the server's main thread by calling **lwp**_setstkcache( ). Then, it enters the dispatching loop. Within the dispatching loop, it first calls **select**( ) to check all the registered sockets for any incoming requests. If there is no request available, it will block, and yield the control to one of the eligible threads. If a new valid request does show up (see the default case within the switch statement), it will create a new thread by calling **lwp**_**create**( ) and set the initial program counter of this thread as svc_getreqset( ). To make remote procedures finish their job faster, it lower the main thread's priority and calls **lwp**_**yield**( ) to pass the control to other non-blocking threads. After getting the control, the new thread retrieves the request message, unmarshals the parameters and calls the desired procedure. Then, the service can be undertaken. Finally, the reply will be sent back to the client and the new thread will be terminated automatically.

```
#define   MAXSVC    10
#define   MAXPRIO   10
#define   MINPRIO   1

void svc_run( ) {
   fd_set     readfds = svc_fdset;
   int        size = getdtablesize( );
   thread_t   tid;

   lwp_setstkcache(1024, MAXSVC + 1);
   while(1) {
     switch(select(size,&readfds,NULL,NULL,NULL)) {
       case -1 :
         ...
       default :
         if(!bcmp(readfds, svc_fdset, sizeof(fd_set)))
           continue;
         lwp_create(&tid, svc_getreqset, MINPRIO, 0,
                    lwp_newstk( ), 1, readfds);
         lwp_setpri(SELF, MINPRIO);
         lwp_yield(tid);
         lwp_setpri(SELF, MAXPRIO);
     }
   }
}
```

Figure 3: The new svc_**run**( ) for concurrent server stubs

3.2 Synchronizing Accesses to Shared Resources

Threads within a server program share many resources, such as variables, files, etc. Therefore, accesses to these shared resources must be synchronized to avoid race condition. This synchronization process is called concurrency control. Because concurrency control is provided by PROTOOL automatically and not aware by programmers, it has to guarantee the property of serializability of remote procedure execution[6]. It employs two-phase locking protocol[6] to guarantee this property. However, two-phase locking protocol may cause remote procedures to deadlock. If such situation occurred within a transaction processing system, it will abort one or more transactions at run

time to break tie. However, in an RPC server, there is no way to abort a remote procedure and recover everything modified by it. Thus, we apply a compile-time deadlock prevention scheme here. Each remote procedure acquires locks on shared items in a predefined order known at compile-time, then deadlock can be prevented[4].

To implement the scheme mentioned above, we employ two components within PROTOOL, i.e., the shared resource analyzer and the source code modifier. They interact like a simple two-phase compiler. At the first phase, the shared resource analyzer recognizes shared variables accessed by each procedure on a server, and builds a procedure activation graph on the server side. In this version of PROTOOL, it recognizes only global and static variables that have no aliases. In next version, we will enhance this component to handle other shared resources.

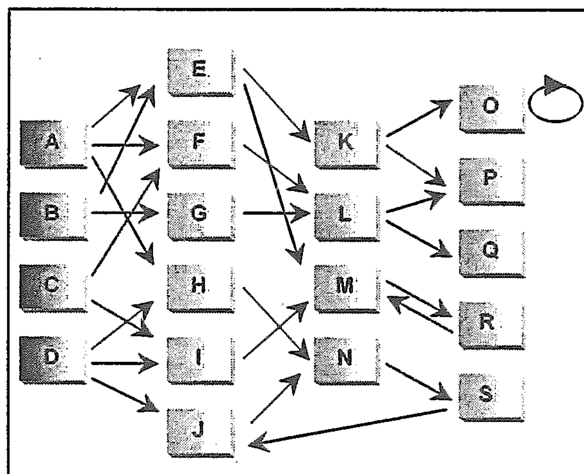

Figure 4: An Example of Procedure Activation Graph

The procedure activation graph contains three kinds of procedures: Procedures belong to the first kind are **boundary procedures** (dark boxes in figure 4). Only remote clients can activate them. Procedures of the second kind are **internal procedures** that have procedure definition in server's source files (boxes of light color in figure 4). They are invisible to clients. In addition, they can activate each other without any constraint, even activate themselves. This implies that there may be direct or indirect procedure recursion. **Library procedures** are of the third kind. They have no procedure definition. Because of the lack of source codes, there is no way to judge which variables within these libraries are shared by these library procedures. Therefore, we try to divide these library procedures into some independent groups in advance. Among procedures within different independent groups, there are no shared resources. The PROTOOL provides some default library procedure groups. Programmers can provide their customized group membership through server configuration file. You can imagine procedures within a library procedure group shared a "big" variable. As long as you activate one of these procedures, you will access this big variable once. Hence, the problem of library procedure activation can be reduced to the problem of accessing shared variable. In figure 4, procedure activation is denoted by a directed link. Any procedures within a server are prohibited to activate remote procedures in another server.

Take procedures in figure 4 as an example. Sets a,b,...,s denote shared variables that can be accessed by procedures A,B,...,S directly (library procedure activation is regarded as an access to a shared variable). Sets a',b',...,s' denote shared variables that can be accessed by procedures A, B,...,S directly or indirectly through procedure activation. What the source code modifier want to know are shared variables that boundary procedure can accessed directly or indirectly. In this example, they are set a', b', c', d'. We propose an algorithm in figure 5 to get them. It traverses the procedure activation graph from each boundary procedure. Its kernel is a function called **compute_accessed( )** that gathers the shared variables accessed directly or indirectly by an internal procedure. More precisely, **compute_accessed(*z*, *path*)** collects the shared variables accessed directly or indirectly by internal procedure *z*. If *z* is an end node in procedure activation graph, it will return the set of shared variables accessed by *z* directly. If *z* is a procedure presented in the ***path*** of the traverse, it will return a null set. (A path is a set of procedures. It shows the activation trail from a boundary procedure to the procedure currently traversed.) If there are still some internal procedures can be activated by *z*, **compute_access( )** will recursively trigger itself to collect the sets of shared variables that can be directly or indirectly accessed by these procedures. Then, it returns the union of these sets of shared variables to its caller.

At the second phase, the source code modifier first collects all shared variables accessed by boundary procedures. Then, it allocates a lock for each accessed shared variables and gives these locks a sequential order. It knows how these boundary procedures access shared variable. Hence, it can insert codes that acquire locks for corresponding shared variable at the beginning of each boundary procedure. Note that each boundary procedure will acquire locks in the sequential order given by the source code modifier. Consequently, deadlock among boundary procedures can be prevented. In addition, it also inserts codes that release all holding locks at the end of each boundary procedure. Therefore, the serializability of remote procedure execution can be guaranteed.

```
SET_VAR s[#_ BOUNDARY_PROCEDURE];

for(each boundary procedure x) {
  s[x] = φ;
  for(each internal procedure y that can be activated by x)
    s[x] = s[x] ∪ compute_accessed(y, φ ∪ x);
}

SET_VAR compute_accessed(PROC z, SET_PROC path) {
  SET_VAR sz = φ;

  if(z is already presented in path)
    return( φ );
  else if(z does not activate any procedures
         defined on server's source file)
    return(The set of shared variables
           accessed directly by z);
  else {
    for(each internal procedure t that can be activated by z)
      sz = sz ∪ compute_accessed(z, path ∪ z);
    return sz;
  }
}
```

Figure 5: An Algorithm to Collect Shared Variables Accessed by Each Boundary Procedure

3.3 Initialization of Servers

Initializing servers is sometimes necessary, such as for some files or for shared variables, and requires programmers to modify server stubs manually. We provide an elegant solution such that programmers only have to collect the codes for initializing the server into some initialization functions, and register them in the server configuration file. Accordingly, the codes for activating these initialization functions would be inserted into the server's main( ) function by the source code modifier. Then, these functions will be executed automatically before dispatching any remote procedure calls from clients.

4. Performance Measurements

In this section, we measure the performance of our concurrent server on SunOS environment. We take the dictionary program in Chapter 22 of Comer's book [3] as the test example, and use two different ways to handle the synchronization problem of the server. One is let PROTOOL to generate synchronization codes automatically, and named it "automated method." Another is to add synchronization codes by hand, and it is called "handcrafted method." We will compare the performance of these two kinds of servers.

We choose TCP as network protocol; start measurements with two clients coexisted, and increase the number of clients in turn until ten. There are three operations in the dictionary program: insertion, deletion, and lookup. Our client first inserts 5000 different strings to server's database, then lookups any 4500 strings out of the 5000 ones, and finally deletes these 4500 strings. After that, we compute the elapsed time for each set of operations (three sets, totally). We repeat the same process for ten times and compute the average time of each set of operations. Table 1 to Table 3 present these results in seconds

Client # / model	2	3	4	5	6	7	8	9	10
Automated Method	1.02	1.03	1.04	1.05	1.08	1.10	1.15	1.21	1.36
Handcrafted Method	1.02	1.03	1.04	1.05	1.06	1.07	1.11	1.16	1.21

Table 1: Measurements of the Insert Operation for Different Servers

Client # / Model	2	3	4	5	6	7	8	9	10
Automated Method	1.03	1.05	1.06	1.09	1.10	1.15	1.19	1.27	1.44
Handcrafted Method	1.03	1.03	1.04	1.05	1.06	1.07	1.11	1.16	1.21

Table 2: Measurements of the Lookup Operation for Different Servers

Client # / Model	2	3	4	5	6	7	8	9	10
Automated Method	1.03	1.04	1.05	1.06	1.09	1.16	1.24	1.43	1.58
Handcrafted Method	1.03	1.03	1.04	1.05	1.07	1.11	1.15	1.25	1.31

Table 3: Measurements of the Delete Operation for Different Servers

The measurements show that the handcrafted server is efficient than the automatically generated one. However, how much superior is it? The results show that the differences are all within 20% and we can conclude that generating synchronization codes automatically is feasible.

5.Conclusion

Complicated software development is the biggest drawback of distributed systems. The PROTOOL provides a way to overcome this troublesome problem. It can handle an application that is comprised of one or more clients and a server at once, and creates a more powerful and efficient server. Users of PROTOOL need not to know any syntax about Sun RPC, and the only thing they must keep in mind is how to separate a program into clients and a server. In contrast, the rpcgen of Sun RPC can handle only one client and one server at once, and generate only inefficient iterative servers. Users of rpcgen must learn how to write interface specifications and interface procedures. This will be a trouble if users are not familiar with them.

Concurrent servers introduce the problem of synchronizing accesses to shared resources. The

PROTOOL uses a deadlock-free approach to deal with this problem and modifies the source files for users automatically. The measurement shows that servers generated by the PROTOOL perform well compared with the server that contains handcrafted synchronization codes.

In the next version of PROTOOL, we will try to enhance our shared resource analyzer to handle shared variables with aliases, and augment it to deal with other shared resources such as files.

References

[1] A.D. Birrell and B.J. Nelson, "Implementing Remote Procedure Calls," *ACM Trans. on Computer Systems*, vol.2, pp.39-59, 1984.

[2] B.D. Marsh, M.L. Scott, T.J. Leblanc, and E.P. Markatos, "First-Class, User-Level Threads," *Proc. of Thirteenth Symp. on Operating Systems Principles*, ACM, pp.110-121, 1991.

[3] Douglas E.Comer, *Internetworking with TCP/IP Vol. III: BSD socket version*, 2nd Ed., pp.277-313, Prentice-Hall, 1996.

[4] G. Coulouris, J. Dollimore, and T. Kindberg, *Distributed Systems – Concepts and Design*, Addison-Wesley Publishing Company Inc., 1994.

[5] G.R. Andrews and F.B. Schneider, "Concepts and Notations for Concurrent Programming," *Computing Surveys*, vol. 15, pp. 3-43, March 1983.

[6] J. Gray and A. Reuter, *Transaction Processing – Concepts and Techniques*, Morgan Kanfmann Publishers, 1993.

[7] Sun Microsystems, Inc., *Network Programming Guild*, March 1990.

[8] Sun Microsystems, Inc., *Programming Utilities & Libraries: Light Weight Processes*, pp.17-47, March 1990.

[9] U. Shani and I. Gold, "Distributed-application Development Tools for DCE/OSF", *Proc. of First International Workshop on Services in Distributed and Networked Environments*, pp. 34-41, 1994.