ELSEVIER

Computer Communications 19 (1996) 528–538

computer communications

Supporting fault-tolerant and open distributed processing using RPC

Wanlei Zhou*

School of Computing and Mathematics, Deakin University, Geelong, VIC 3217, Australia

Received 28 October 1994; revised 2 April 1995

Abstract

This paper is concerned mainly with the software aspects of achieving reliable operations in an open distributed processing environment. A system for supporting fault-tolerant and cross-transport protocol distributed software development is described. The fault-tolerant technique used is a variation of the recovery blocks and the distributed computing model used is the remote procedure call (RPC) model. The system incorporates fault tolerance features and cross-transport protocol communication features into the RPC system and makes them transparent to users. A buddy is set up for a fault-tolerant server to be its alternative. When an RPC to a server fails, the system will automatically switch to the buddy to seek for an alternate service. The client, the fault-tolerant server and the buddy of the server can all use a different transport protocol. To obtain this fault tolerance and cross-protocol service, users only need to specify their requirements in a descriptive interface definition language. All the maintenance of fault tolerance and the cross-protocol communication is managed by the system in a user transparent manner. By using our system, users will have confidence in their distributed programs without bothering the fault tolerance and cross-protocol communication details. Our system is small, simple, easy to use and also has the advantage of producing server and client driver programs, and finally, executable programs directly from the server definition files.

Keywords: Open distributed processing; Fault-tolerant computing; Distributed systems; Remote procedure calls; Client/server model

1. Introduction

The advances in computer technology have made it cost-effective to build distributed systems in various applications. Many experts agree that the future of distributed computing, especially the future of open distributed processing, is the future of computing. 'The network is the computer' has become a popular phrase [1].

Remote Procedure Call (RPC) is perhaps the most popular model used in today's distributed software development, and has become a de facto standard for distributed computing. To use it in an open distributed environment effectively, however, one has to consider cross-protocol communications, because user programs built on top of different RPC systems cannot be interconnected directly. Typical solutions to this problem are:

(1) Black protocol boxes: protocols used by RPC programs are left as black boxes in compiling time, and are dynamically determined in binding time [2].
(2) Special interfaces [3] or RPC agent synthesis systems [4] for cross-RPC communications.

However, one issue is still outstanding in building RPC systems for open distributed systems; fault-tolerance features.

An open distributed system consists of many hardware/software components that are likely to fail eventually. In many cases, such failures may have disastrous results. With the ever increasing dependency being placed on open distributed systems, the number of users requiring fault tolerance is likely to increase.

The design and understanding of fault-tolerant open distributed systems is a very difficult task. We have to deal with not only all the complex problems of open distributed systems when all the components are well, but also the more complex problems when some of the components fail.

* Email: wanlei@deakin.edu.au

PII S0140-3664(96)01079-1

This paper is concerned mainly with the software aspects of achieving reliable operations in an open distributed processing environment. A system for supporting fault-tolerant and cross-transport protocol distributed software development is described. The system design is aimed towards application areas that may involve a heterogeneous environment, and in which requirements for fault-tolerance are less severe than in, for example, the aerospace field, but in which continuous availability is required in the case of some components failures [5]. The application areas could be, for example, kernel/service pool-based distributed operating systems, supervisory and telecontrol systems, switching systems, process control and data processing. Such systems usually have redundant hardware resources, and one of the main purpose of our system is to manage the software redundant resources in order to exploit the hardware redundancy. The fault-tolerant technique used is a variation of the recovery blocks technique, and the distributed computing model used is the RPC model.

Software fault tolerance refers to the set of techniques for continuing service despite the presence, and even the manifestation, of faults in a program [6]. There are many techniques available for software fault-tolerance, such as N-version programming [7] and recovery blocks [8]. In N-version programming, $N(N \geq 2)$ independent and functionally equivalent versions of a program are used to process a critical computation. The results of these independent versions are compared (usually with a majority voting if N is odd) at the end of each computation, and a decision will be made accordingly.

Recovery blocks employ temporal redundancy and software standby sparing [9]. Software is partitioned into several self-contained modules called recovery blocks. Each recovery block consists of a *primary routine*, which executes critical software function; one or more *alternate routines*, which performs the same function as the primary routine, and is invoked upon a failure is detected; and an *acceptance test*, which tests the output of the primary (and alternate, if the primary fails) routine after each execution. A variation of this model is used in this paper.

The remote procedure call is a powerful and widely known primitive for distributed programming [10]. The RPC based model allows a programmer to call a procedure located at a remote computer in the same manner in which a local procedure is called. This model has a lot of advantages. The procedure call is a widely accepted, used and understood abstraction. This abstraction is the sole mechanism for accessing remote services in this model. So the interface of a remote service is easily understood by any programmer with a good knowledge of ordinary programming languages.

The remainder of this paper is organised as follows. In Section 2, we summarize some notable related work and provide the rationale of our work. In Section 3, we describe the architecture of the SRPC system. Then Section 4 describes the syntax and semantics of the server definition files and the stub and driver generator. In Section 5 we present an example to show how this system can be used in supporting fault-tolerant, open distributed software development. Section 6 contains remarks.

2. Related work and the rationale

There have been many successful RPC systems since Nelson's work [11], such as Cedar RPC [12], NCA/RPC [13], Sun/RPC [14], HRPC [2], and so on. But few of them consider fault tolerance an cross-protocol communication in their design, or they rely on users to build in these features.

Notable work on incorporating fault tolerance features into RPC systems is Argus [15], ISIS [16,17] and an atomic RPC system on ZMOB [18]. The Argus allows computations (including remote procedure calls) to run as atomic transactions to solve the problems of concurrency and failures in a distributed computing environment. Atomic transactions are serializable and indivisible. A user can also define some atomic objects, such as atomic arrays and an atomic record, to provide the additional support needed for atomicity. All the user fault tolerance requirements must be specified in the Argus language.

The ISIS toolkit is a distributed programming environment, including a synchronous RPC system, based on virtually synchronous process groups and group communication. A special process group called a fault tolerant group, is established when a group of processes (servers and clients) are cooperating to perform a distributed computation. Processes in this group can monitor one another, and can then take actions based on failures, recoveries or changes in the status of group members. A collection of reliable multicast protocols is used in ISIS to provide failure atomicity and message ordering.

The atomic RPC system implemented on ZMOB uses sequence numbers and calling paths to control the concurrency and atomicity, and used checkpointing to maintain the ability of recovering from failures. Users do not have to provide synchronization and recovery themselves; they only need to specify if atomicity is desired. This frees them from managing much complexity.

But when a server (or a guardian in the Argus) fails to function well, an atomic transaction or an atomic RPC has to be aborted in these systems. This is a violation of our continuous computation requirement. The fault-tolerant process groups of the ISIS can cope with process failures and can maintain continuous computation, but the ISIS toolkit is big and relatively complex to use.

Typical solutions to the cross-protocol communication in RPC systems are the black protocol boxes of the HRPC [2], the special protocol conversion interface [3] and the RPC agent synthesis system [4] for cross-RPC communications.

The HRPC system defines five RPC components: the stub, the binding protocol, the data representation, the transport protocol and the control protocol. An HRPC client or server and its associated stub can view each of the remaining components as a 'black box'. These black boxes can be 'mixed and matched'. The set of protocols to be used is determined at bind time–long after the client and server has been written, the stub has been generated, and the two have been linked.

The special protocol conversion interface that we proposed earlier [3] uses an 'interface server' to receive a call from the source RPC component (client or server) and to convert it into the call format understood by the destination RPC component (server or client).

The cross-RPC communication agent synthesis system [4] associates a 'client agent' with the client program and a 'server agent' with the server program. A 'link protocol' is then defined between the two agents and allows them to communicate. The server and the client programs can use different RPC protocols, and the associated agents will be responsible of converting these dialect protocols into the link protocol.

But none of the above cross-protocol RPC systems consider fault-tolerance issues. If the server fails, the client simply fails as well.

Incorporating both fault tolerance and cross-protocol communication into RPC systems is clearly an important issue to use RPCs efficiently and reliably in open distributed environments. In this paper we describe a system, called the SRPC (Simple RPC) system, for supporting the development of fault-tolerant, open distributed software. The SRPC incorporates fault tolerance features and protocol converters into the RPC system and makes them transparent to users. A *buddy* is set up for a fault-tolerant server to be its alternative. When an RPC to a server fails, the system will automatically switch to the buddy to seek for an alternate service. The RPC aborts only when both the server and its buddy fail. The clients and servers can use different communication protocols. To obtain these fault tolerance and automatic protocol converting services, users only need to specify their requirements in a descriptive interface definition language. All the maintenance of fault tolerance and protocol conversion are managed by the system in a user transparent manner. By using our system, users will have confidence on their open distributed computing without bothering the fault tolerance details and protocol conversion. Our system is small, simple, easy to use and also has the advantage of producing server and client driver programs and finally executable programs directly from the server definition files.

3. System architecture

The SRPC is a simple, fault-tolerant and cross-protocol remote procedure call system [19]. The system is small, simple, expandable, and it has facilities supporting fault-tolerant computing and cross-protocol communication. It is easy to understand and easy to use. The SRPC only contains the essential features of an RPC system, such as a location server and a stub generator, among other things. The SRPC system has been used as a distributed programming tool in both teaching and research projects for three years.

The SRPC system has another interesting feature. That is, the stub compiler (we call it the stub and driver generator, or SDG in short) not only produces the server and client stubs, but also creates remote procedures' framework, makefile, and driver programs for both server and client. After using the `make` utility, a user can test the program's executability by simply executing the two driver programs. This feature will be more attractive when a programmer is doing prototyping.

3.1. Server types

The client/server model [20] is used in the SRPC system. An SRPC program has two parts: a server part and a client part. Usually the server provides a special service or manages an object. The client requests the service or accesses the object by using the remote procedures exported by the server.

There are three types of servers in the SRPC system: simply servers, service providing servers and object managing servers. Fig. 1 depicts these three types of servers.

A simple server (Fig. 1(a)) is an ordinary server possessing of no fault-tolerant features. When a simple server fails, all RPCs to it have to be aborted.

A service providing server (Fig. 1(b)) has a buddy server running somewhere in the network (usually on a host different with the server's), but no communication between the server and its buddy. When a service providing server fails, an RPC to this server will be automatically redirected to its buddy server by the system. As object changes in the server will not be available in its buddy, a service providing server usually is used in applications such as pure computation, information retrieval (no update), motor-driven (no action memory), and so on. It is not suitable to be used to manage any critical object that might be updated and then shared by clients.

An object managing server (Fig. 1(c)) also has a buddy running in the network. It manages a critical object that might be updated and shared among clients. An RPC to such a server, if it will change the object state, is actually a nested RPC. That is, when the server receives such a call from a client, it first checks to see whether the call can be

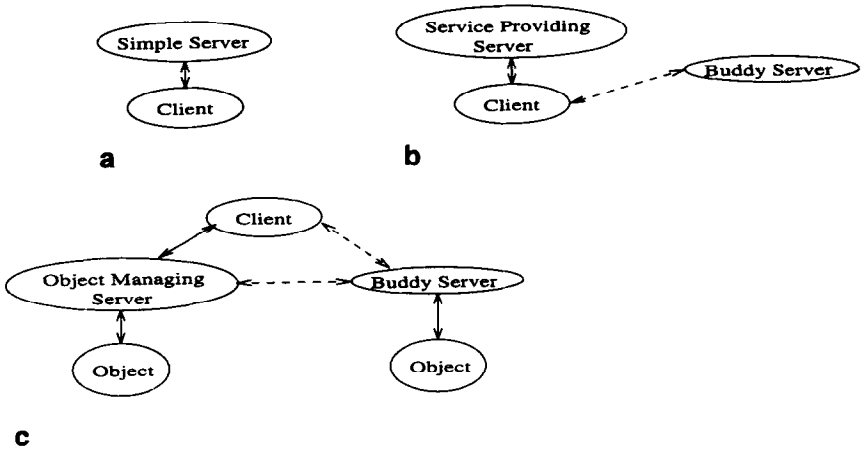

Fig. 1 Server types.

executed successfully (e.g. if the necessary write-locks have been obtained or not). If the answer is no, the call is aborted. If the answer is yes, then the server will call its buddy server to perform the operation as well. When the buddy returns successfully, the call commits (the server and its buddy actually perform the call) and the result returns to the client. To ensure the consistency of the objects managed by the server and its buddy, a two-phase commit protocol [21] is used when executing the nested RPC.

Like a service providing server, when an object managing server fails, an RPC to this server will be automatically redirected to its buddy server by the system.

All buddy servers are simple servers. That means, when a server (service providing or object managing) fails, its buddy server provides alternative service in a simple server manner. Also, when a buddy server fails, a service providing server or an object managing server will be reduced into a simple server.

3.2. Architecture

The SRPC has the following three components: a Location Server (LS) and its buddy (LS buddy), a system library, and a Stub and Driver Generator (SDG). This section describes the system architecture from a user's point of view. As server buddies are generally transparent to users, we will omit their descriptions here.

From a programmer's viewpoint, after the SDG compilation (see Section 5), the server part of an SRPC program is consisted of a server driver, a server stub and a file which implements all the remote procedures (called a procedure file). The server buddies are transparent to users. The server part (or a server program as it is sometimes called) is a 'forever' running program which resides on a host and awaits calls from client. The client part (or a client program) consists of a client driver and a client stub after the SDG compilation. It runs on a host (usually a different host from the server's host), and makes calls to the server by using the remote procedures exported by the server.

When the client driver makes a call, it goes to the client stub. The client stub then, through the system library, makes use of the client protocol for sending the calling message to the server host. Because the client and the server may use different communication protocols, a client-server protocol converter is used to convert the client's protocol into server's protocol. The calling message is then sent to the server. At the server's host side, the server's protocol entity will pass the calling message to the server stub through the system library. The server stub then reports the call to the server driver and an appropriate procedure defined in the procedures file is executed. The result of the call follows the calling route reversely, through the server stub, the server protocol, the system library of the server host, the client-server protocol converter, the system library of the client host, the client stub, back to the client driver. This is called a direct call, as the pre-condition of such a call is that the client knows the address of the server before the call.

With the help of the Location Server, the run-time address of a server can be easily accessed. One typical scenario of SRPC programs using LS can be described below:

1. Registering: when the server is started, it first registers its location to the LS. The route is: server driver → server stub → server protocol, server-LS protocol converter and the system library of the server host → .LS protocol and system library of the LS host → LS stub → LS driver.
2. Waiting: the server waits for client calls.
3. Locating: when a client is invoked, it consults the LS for the server's location. The route is: client driver → client stub → client protocol, client-LS protocol converter and system library of the client host → LS protocol and system library of the LS host → LS stub → LS driver.

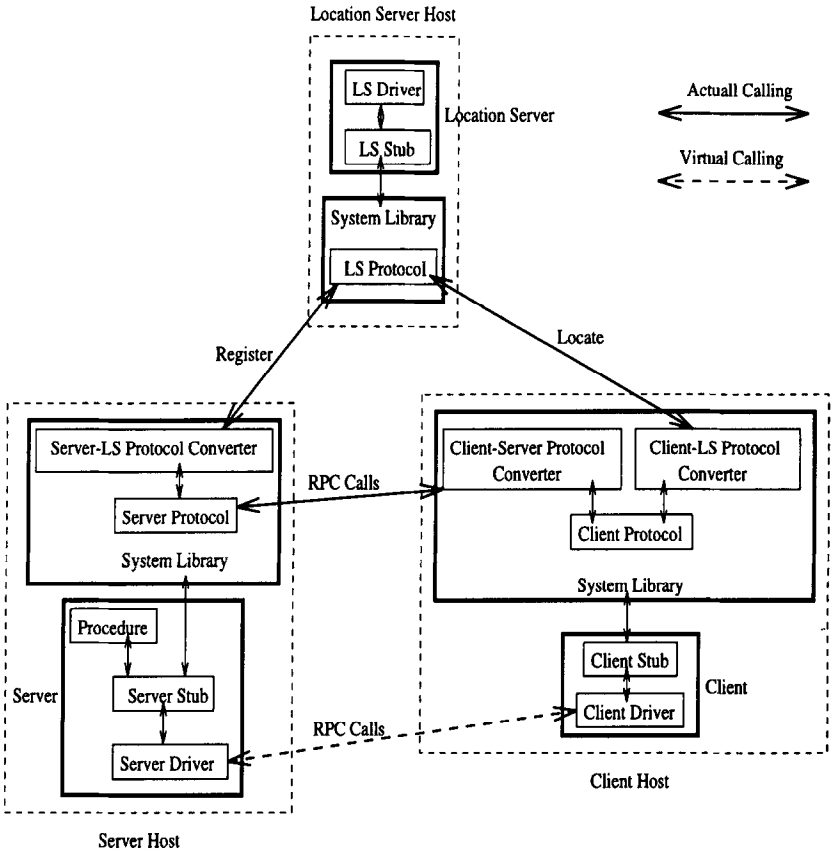

Fig. 2. System architecture and a typical RPC. ⟷: Actual calling; ⇜ - - ⇝: virtual calling.

4. RPC: after the location is found, the client then can make any number of RPCs to that server by using the obtained location (as in a direct call). We name this a *typical calling*, since most of the time a client does not know server addresses.
5. Shutdown: if a 'shutdown' call is issued by a client program, it causes the server to un-register itself from the LS and exits from the system.

Fig. 2 depicts the system architecture using a typical RPC. The dashed line represents the RPCs from the user's viewpoint.

3.3. Location server

One way of hiding out the implementation details is the use of the Location Server (LS). The LS is used to hide the server locations from users. It maintains a database of server locations and is executed before any other SRPC program is started. After that, it resides on the host and awaits calls from servers and clients.

The Location Server is an object managing server and has a buddy of its own. It has a well-known location, and this location can be easily changed when necessary. The LS itself is implemented by the SRPC system, using the direct calling method.

Usually there should be one LS (called local LS) running on each host for managing locations of that host, and these local LSs report to the 'global LS' (like the NCA/RPC's local and global location brokers) [13,22]. In that case, the locations of all LSs can also be hidden from users. We have planned to implement this facility.

The following call is used by a server to register itself to the LS:

```
int registerServer(sn, buddy, imp)
char *sn;          /* server name */
char *buddy;       /* buddy's name */
struct iinfo       /* implementation
*imp;                 info. */
```

where `imp` is a type `struct iinfo` structure and contains many implementation details, such as the server's host name, protocol, and so on. Because the call updates the LS database, it is also directed to the LS buddy. If the call returns OK, the location has been registered and a client can use the following call to find out the location of a server from the LS:

```
int locateServer(sn, buddy, imp)
char *sn;          /* server name */
char *buddy;       /* server's buddy
                      name */
struct iinfo       /* implementation
*imp*;             info. */
```

If the call returns OK, the location of the server sn is stored in imp and the name of the server's buddy is stored in buddy for later use. This call does not affect the LS database state, so there is no hidden LS server and LS buddy communication here. Before a server is shut down, the following call must be used to un-register the server from the LS:

```
int unregisterServer(sn)
char *sn;          /* server name */
```

If the call returns OK, the server and its buddy (if any) are deleted from the LS database. The system also provides other LS calls for maintaining the LS database.

All the usages of these functions in a server or a client program are automatically generated by the stub and server generator. A user does not need to look into the details of these calls if he or she is satisfied with the generated program sections.

3.4. System library

The system library is another way of achieving transparency. The library contains all the low-level and system- and protocol-oriented calls. Its main function is to make the low-level facilities transparent to the upper-level programs. So the stub and driver programs of both server and client will not deal with their communication entities directly.

The server and client programs must be linked with the system library separately. Ref. [19] contains detailed descriptions of the library calls. All the library calls can be divided into the following call levels, and Fig. 3 depicts their relationships:

1. SRPC Level: this is the highest level. It contains calls that deal with RPC-related operations.
2. Remote Operation Level: contains calls that deal with remote operations. These remote operations follow the definitions of the OSI Application level primitives [23].
3. Protocol Level: contains calls that deal with protocol-specific operations.
4. Utility calls: contains all the utility calls used in different levels.

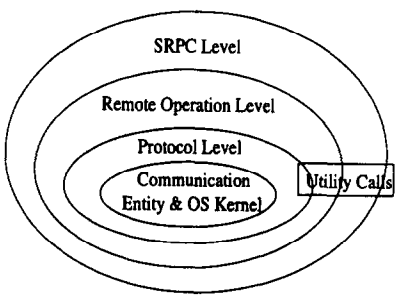

Fig. 3 Relationships of system library levels.

The inner most level is the protocol-specific level. It interfaces with the specific protocol entity and the underlying operating system kernel. It is also responsible for providing protocol converting procedures. The remote operation level provides a uniform interface (similar to the OSI Application level primitives) to the upper RPC system. The uniform interface provides two obvious advantages:

- It provides a clear interface for different communication protocols.
- It makes the SRPC system as portable as possible: only the lower levels need to be re-programmed when port the SRPC system to other platforms.

The SRPC level implements all the RPC related calls and provides a user-friendly remote procedure calling interface to application programs. The utility calls provide service calls for different levels.

3.5. Performance evaluation

Obviously, our service providing server and object managing server can tolerant single-point failures. This is clearly an advantage compared with the normal single server model. However, every fault-tolerant method employs redundancy, and redundancy may imply increasing costs and/or decreasing performance. Similarly, protocol conversion also involves system overhead. This section tries to evaluate the performance of our server types in various circumstances.

The performance of an RPC in the SRPC system varies, according to which server type is used. Table 1 lists the null RPC performance on a network of HP and Sun workstations, where the server program runs on an HP 715/33 workstation and the server buddy and the client run on two separate Sun 4/75 ELC (33 MHz) workstations. The network is virtually isolated and very lightly loaded (no other user programs were run except the testing programs during data collection). The server (and the buddy, of course) uses the Internet_datagram (UDP) protocol and the client uses the Internet _stream (TCP) protocol.

The table also includes null RPC times in the SRPC system for simple servers without protocol conversion. That is, when both the client and server use the same protocol (UDP or TCP), and therefore, no protocol conversion is required.

We can draw the following observations and explanations from Table 1:

- The overhead of protocol conversion is light. From the table we know the average simple RPC time using UDP and TCP protocol only is about 2.59 ms. The simple RPC protocol conversion uses about 3.22 ms, which is only 0.63 ms more than the time used by simple RPCs without protocol conversion. When

Table 1
Null RPC performance

Server type	Time (ms)
Simple (UDP)	2.11 ± 0.02
Simple (TCP)	3.07 ± 0.02
Simple	3.22 ± 0.02
Service-providing	3.37 ± 0.02
Object-managing	5.12 ± 0.04

a network is normally loaded, a null RPC time typically needs 5–10 ms. In that case, the extra time for protocol conversion could be less than 10% of the RPC time.

- The overhead of using service providing server is minimum. From the table we can see that the time difference between a simple RPC and a service providing RPC is only 0.15 ms. Most of the extra time is spent on the preparation of using the buddy server in case of server failure.
- The overhead of using object managing server is quite high. This is because of the nested RPC used in keeping the consistency between the two objects managed by the server and the buddy. However, the time used is less than the time of two simple RPCs. This is because that there is no protocol conversions between the server and its buddy, and some of the operations are carried out in parallel.

We are still investigating the system performance under other circumstances, such as RPCs with various sizes of parameters and with various network load conditions.

4. Stub and driver generator

4.1. Syntax

The purpose of the stub and driver program generator is to generate stubs and driver programs for server and client programs according to the Server Definition Files (SDF). The syntax of a server definition file is shown in Listing 1.

We use a modified BNF to denote the syntax of definition files. The 'variable', 'integer', 'string', 'constant', and 'declarator' have the same meanings as in the C programming language. Comments are allowed in the definition file. They are defined the same as in the C programming language using /* and */).

4.2. Semantics

Most of the descriptions of Listing 1 are self-explanatory. We only highlight the following points:

1. The server's name is defined as a variable in the C language. This name will be used in many places.

```
<SDF>::=          BEGIN
    <HEADER>
    [<CONST>]
    <FUNCS>
    END

<HEADER>::=       Server Name:
    variable ;
    Comment: string ;
    Server Protocol:
    variable ;
    Client Protocol:
    variable ;
    [<BUDDY>]

<BUDDY>::=        Buddy <BDYTYPE>:
    variable ;
    Using: <LANGUAGE> ;
<BDYTYPE>         ::= Auto | Forced
<LANGUAGE>        ::= C | Pascal
<CONST>           ::= constant
<FUNCS>           ::= RPC Functions:
    <RPCS>
<RPCS>            ::= <RPC> { <RPC> }
<RPC> ::= Name: string
    [Update] ; <PARAMS>
<PARAMS>          ::= { <PARAM> }
<PARAM>           ::= Param: <CLASS>:
    declarator;
<CLASS>           ::= in | out
```

Listing 1. Server definition file syntax.

For example, it is the key in the LS database to store and access server entities. When the client asks the LS to locate a server, it provides the server's name defined here. The name is also used as a prefix in naming all the files generated by the SDG. So two different servers cannot be assigned to the same server name. Otherwise, the server who registers to the LS first will be accepted while the server who registers to the LS later will be rejected by the LS.

2. Different protocols can be defined for the server and the client, respectively. The buddy, if it is defined, uses the same protocol as the server does. Currently, only three protocols are allowed: Internet_datagram (The UDP protocol), Internet_stream (the TCP protocol), and XNS_datagram (the XNS packet exchange protocol).
3. The <BUDDY> part is optional. If it is not specified, the generated server will be a simple server, otherwise it will be a service providing server or an object managing server, according to some definitions in the <RPCS> part (described below). The <BUDDY> part has a buddy definition and a language definition. The buddy definition defines that whether the buddy's name and execution is to be determined by the system

(Auto) or to be determined by the programmer (Forced). If Auto is defined, the system will generate the buddy server's name (ServerNameBdy, used for registering and locating it), the buddy's driver and stub files as well as the *makefile*, and will treat the following variable as the name of the buddy's procedure file. Then, the buddy program will be compiled and executed together with the server program. The host of the buddy program will be determined by the system at run time.

If Forced is defined, the generator will not generate any buddy's program file and will treat the following variable as the name of the buddy server used for registering and locating. The programming and execution of the buddy server will also be the programmer's responsibility.

The language definition Using defines which language does the buddy program use. The key issue of software fault-tolerant is the *design diversity* or version independent, and one way of achieving design diversity is through the use of multiple programming languages [24]. If a different language is chosen for each version implementation, then the versions are likely to be more independent, not only due to the diversity of languages, but also because individual language features force programmers toward different implementation decisions. Currently only the C programming language is supported in the SRPC system. We have planned to support the Pascal language implementation soon.

4. The <FUNCS> part defines the remote procedures of the server. At least one remote procedure must be defined. Each remote procedure is defined as a name part and a parameter (<PARAMS>) part. The name of a remote procedure is simply a variable, with an optional Update definition. The latter definition distinguishes an object managing server with a service providing server. That is, if the <BUDDY> part is defined and the Update is defined in any one RPC definition, the server is an object managing server. If only the <BUDDY> part is defined but no Update part is defined in any RPC definition, the server is a service providing server. The meaning of the Update definition is: if an Update is defined following an RPC procedure name, that procedure must be maintained as a nested RPC affecting both the server and the buddy by the server program (see Section 3.1).

There can be zero or several parameters in a procedure, each consisting of a class and a declaration. The class can be in or out, which tells the SRPC system that the parameter is used for input or output, respectively. The declaration part is the same as in the C language. In this version, only simple character string is allowed in parameter definitions. Further extensions are under way.

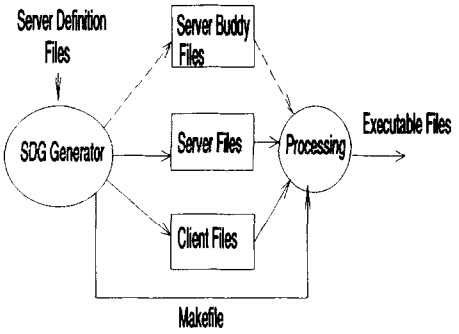

Fig. 4 Processing structure of the stub and driver generator.

4.3. Implementation issues

After a programmer sends a server definition file to the generator, the generator first does syntax checking. If no errors are found, several program source files and a makefile are generated. The subsequent processing is specified by the makefile. That is, when using the make utility, the executable files of both the server and client will be generated. Fig. 4 indicates the structure of the processing. The dashed lines represent optional actions.

At least one server definition file must be input to the SDG. If there are more than one server, their SDFs can be input to the SDG simultaneously. If there is only one SDF file, then the generated client driver can execute the server's procedures one by one. If the buddy part is also specified, the generated client can also call the buddy procedures directly (this is useful in testing the client-buddy communication).

If there are more than one SDF file, then for each server, the SDG will generate one set of server files, one set of client files, and one set of buddy files (if the buddy is defined), respectively. These files are the same as the servers being processed in single file input described above. One additional set of client files, the *multi-server client* program, will also be generated in this case. The client driver is called a *multi-server client driver*. It can call all the procedures of all the servers one by one. A further improvement is under way to let the client call these procedures in parallel.

5. Application example

We use a simple example to show the application of the SRPC system. Suppose we have a server definition file called sf.def. It defines a 'send-and-forward' system in that the server acts as a message storage and the client acts as both message sender and receiver. The server definition file is shown in Listing 2.

From the header part of this SDF file we know the following: the server is named as sf and the server protocol used is the Internet_datagram. The server can

```
/* Store and forward: server
definition file */

BEGIN
    Server Name: sf;
    Comment: Store and forward
    system;
    Server Protocol:
    Internet_datagram;
    Client Protocol:
    Internet_stream;
    Buddy Auto: sfBdyOps.c;
    Using: C;

    #define MXNAML 64
    #define MXMSGL 500
    #define MXSTRL 80

RPC Functions:
    Name: storeMsg Update;
      Param: in receiver: char
      receiver[MXNAML];
      Param: in msg: char
      msg[MXMSGL];
      Param: out stat: char
      stat[MXSTRL];
Name: forwardMsg Update;
      Param: in receiver: char
      receiver[MXNAML];
      Param: out msg: char
      msg[MXMSGL];
Name: readMsg;
      Param: in receiver: char
      receiver[MXNAML];
      Param: out msg: char
      msg[MXMSGL];
Name: listMsg;

END
```

Listing 2. Server definition file example.

be then executed on any host using any port assigned by the system. The client protocol is Internet_stream and can also be executed on any host. Obviously, a protocol converter is needed. A server buddy is defined and is expected to be established automatically by the system, using the Internet_datagram protocol (the same as the server). The procedure file for the buddy is `sfBdyOps.c` and the programming language used for the buddy is the C language. There are also three constants defined.

From the RPC functions part we know that four remote procedures are defined in this SDF file. The first two RPC functions are marked as `Update`. So the server is an object managing server. When the client calls these two procedures, these two procedures will be treated as nested calls for maintaining the object consistency in both the server and its buddy. The next two RPC function definitions have no `Update` marks, and then they will be treated as ordinary RPCs.

When the file is input to the generator, the following files will be generated:

```
sf.h Header file, must be included by server,
     its buddy and client drivers and stubs.
sfSer.c          Server driver file.
sfStubSer.c      Server stub file.
sfOps.c          Frameworks of
                 server procedures.
sfCli.c          Client driver file.
sfStubCli.c      Client stub file.
sfBdy.c          Server buddy driver
                 file.
sfStubBdy.c      Server buddy stub
                 file.
makefile         Make file.
```

After using the make utility (simply use 'make' command), three executable files are created:

```
sfSer            Server program.
sfCli            Client program.
sfBdy            Server buddy
                 program.
```

Note that the `sfOps.c` file only defines the frameworks of the remote procedures (dummy procedures). Their details are to be programmed by the programmer. The `sfBdyOps.c` file should be the same as the `sfOps.c` file (the only possible difference happens when the server buddy uses another programming language such as the Pascal; then the affix of the file would be `.pas`).

The server driver is simple. It does the initialization first, then it registers with the LS and invokes the buddy program on a neighbouring host because the buddy is defined as `Auto` in the SDF file. After that it loops 'forever' to process incoming calls until the client issues a 'shutdown' call. In that case, the server un-registers from the LS and exits. The 'un-register' call will automatically un-register the buddy from the LS as well. The incoming calls are handled by the server stub and underlying library functions. A listing of the server driver is shown in Listing 3.

The server buddy driver works in the same way as the server program, except that it does not invoke a buddy program. Also, the buddy is a simple server and all calls to the buddy will not be nested.

The generated client driver can execute the server's remote procedures one by one. If the server driver is running and the client driver is invoked, the client driver first lists all the remote procedures provided by the server, and asks the user to choose from the list. The following is the menu displayed for this example:

```
Initialisation (including invoke the
buddy);
/* Register the server to the LS */
registerServer("sf", "sfBdy", imp);
while (1) {
   wait for client calls;
   /* comes here only if a client
   called */
   fork a child process to handle
   the RPC;
   if the call is "shutdown"
      break;
}
unregisterServer("sf");
```

Listing 3. Server driver pseudocode.

```
Available calls:

0  sf$Shutdown
1  sf$storeMsg(receiver, msg, stat)
2  sf$forwardMsg(receiver, msg)
3  sf$readMsg(receiver, msg)
4  sf$listMsg()

Your choice:
```

After the selection, the input parameters of the named remote procedure are then input from the keyboard. After that, the driver program does some initialization and the remote procedure is executed and returned results displayed. The actual calling and displaying are handled by the client stub and underlying library functions. The format of all the four RPCs in the client program are the same as the format listed in the above menu. That is, if the client wants to send a message to a receiver, it does the following call after the receiver's name and the message are input into `receiver` and `msg` variables, respectively:

```
sf$storeMsg(receiver, msg, stat);
```

Note that the remote procedure's name is named as a composition of the server's name `sf`, a $ sign, and the remote procedure's name `storeMsg` in the SDF file. Similarly, if the client wants to receive messages, it does the following call after the receiver's name `receiver` is obtained:

```
sf$forwardMsg(receiver, msg);
```

Before each RPC, a `locateServer("sn", buddy, imp)` call is issued to the LS to return the location of the server and the name of its buddy. The server location is stored in `imp` and the buddy name is stored in `buddy`.

The fault-tolerant feature of the system is completely hidden from the user. For this example, all the remote procedure calls from the client program will be first handled by the server. A nested RPC is issued if the incoming call is either `sf$storeMsg(receiver, msg, stat)` or `sf$forwardMsg(receiver, msg)`. This is because the two RPC functions are marked as `Update` in the SDF file. The nested RPC will ensure that actions of the incoming call will be made permanent on both the server and its buddy if the call is successful, and no actions of the incoming call will be performed if the call fails. Two other incoming calls, `sf$readMsg(receiver, msg)` and `sf$listMsg()`, will be handled by the server only.

If the server fails (that is, the RPC to the server returns an error), the client program will send the RPC to the server's buddy. The location of the buddy will be determined by another call to the LS:

```
locateServer(buddy, "", imp)
```

where `buddy` is the server buddy's name obtained during the first call to the LS, and `imp` stores the location of the buddy.

The cross-protocol communication is also hidden from the user. All the interfaces to the protocol converters (client-LS, client-server, and server-LS) are generated by the SDG (in the stub files) and used automatically by the stubs. If a user only deals with the RPC level, he or she will never notice the underlying protocols used by the server and the client programs.

The termination of the server program also needs to be mentioned. After the server program is started, it will run forever unless the programmer kills its process, or there exists a facility to terminate the server. Here we provide a facility to do that job. We add a 'remote shutdown' procedure into the server, and allow the remote shutdown of the server in the server program. Hence, when the client driver calls the remote shutdown procedure of the server, the server will shut down itself and will exit from the system.

6. Remarks

A system for supporting fault-tolerant, open distributed software development is described in this paper. The system is simple, easy to understand and use, and has the ability of accommodating multiple communication protocols and tolerating server failures. It also has the advantage of producing server and client driver programs, and finally executable programs directly from the server definition files. The system has been used as a tool of distributed computing in both third year and graduate level teaching, and has been used by some students in their projects.

In tolerating server failures, similar efforts can be found in the RPC systems that provide replicated server facilities, such as NCA/RPC [13]. But in these systems, the user, instead of the system, takes the responsibility of maintaining and programming the functions for object consistency. This is a difficult job for many programmers.

The Argus system and other systems that maintain transaction atomicity also provide some sort of fault tolerance for servers (guardians in the Argus), but their purpose is to maintain the transaction atomicity, that is, if a server fails the transaction may abort and it has no effect on the accessed objects, and other transactions will not be affected. Our approach in achieving fault tolerance is similar to the approach used in the ISIS toolkit (of course, ours is more simplified and less powerful). But our system is simple, easy to understand and easy to use. In our system, we provide a server buddy to tolerant the server's failure. When the server fails, the client, instead of aborting, can access the server buddy to obtain the alternative service. Also in our system, it is the system, instead of the user, that is responsible of maintaining the consistency of the managed objects.

Providing server and driver programs directly from the server definition file (similar to the interface definition files of other RPC systems) is also an interesting characteristic of our system. It is related to the rapid prototyping of RPC programs [25]. The driver programs are simple, but have the advantages of testing the executability of the RPC program immediately after the designing of the SDF file. It is especially useful if the user makes some changes in the SDF file or the procedure file. In that case, these changes will be automatically incorporated into other related program files if the program is re-generated by the stub and driver generator. This will avoid a lot of trouble in the maintenance of consistency of program files.

References

[1] D. Cerutti, The network is computer, in D. Cerutti and D. Pierson (eds.) Distributed Computing Environments, McGraw-Hill, New York, 1993, pp. 17–26.

[2] B.N. Bershad, D.T. Ching, E.D. Lazowska, J. Sanislo and M. Schwartz, A remote procedure call facility for interconnecting heterogeneous computer systems, IEEE Trans. Software Engineering, 13(2) (August 1987) 880–894.

[3] W. Zhou, A remote procedure call interface for heterogeneous computer systems, Proc. Open Distributed Processing Workshop, Sydney, Australia, January 1990.

[4] Y.-M. Huang and C.V. Ravishankar, Designing an agent synthesis system for cross-rpc communication, IEEE Trans. Software Engineering, 20(3) (March 1994) 188–198.

[5] M. Boari, M. Ciccotti, A. Corradi and C. Salati, An integrated environment to support construction of reliable distributed applications (CONCORDIA), in Parallel Processing and Applications, Elsevier, Amsterdam, 1988, pp. 467–473.

[6] A. Mili, An Introduction of Program Fault Tolerance, Prentice-Hall, Englewoods Cliffs, NJ, 1990.

[7] A. Avizienis, N-version approach to fault-tolerant software, IEEE Trans. Software Engineering, 11(12) (December 1985) 1491–1401.

[8] B. Randell, System structure for software fault tolerance, IEEE Trans. Software Engineering, 1(2) (June 1975) 220–232.

[9] H. Hecht and M. Hecht, Fault-tolerant software, in Fault-Tolerant Computing: Theory and Techniques, Vol. 2, Prentice-Hall, Englewoods Cliffs, NJ, 1986, pp. 658–696.

[10] G.K. Gifford, Communication models for distributed computation, in Distributed Operating Systems, Theory and Practice, NATO ASI Series Vol. F28, Springer-Verlag, Berlin, Germany, 1986, pp. 147–174.

[11] B.J. Nelson, Remote procedure call, Technical Report CSL-81-9, Xerox Palo Alto Research Centre, May 1981.

[12] A.D. Birrell and B.J. Nelson, Implementation remote procedure calls, ACM Trans. Computer Systems, 2(1) (February 1984) 39–59.

[13] L. Zahn, T.H. Dineen, P.J. Leach, E.A. Martin, N.W. Mishkin, J.N. Pato and G.L. Wyant, Network Computing Architecture, Prentice-Hall, Englewoods Cliffs, NJ, 1990.

[14] Sun Microsystems, RPC: Remote procedure call protocol specification version 2 (RFC 1057), in Internet Network Working Group Request for Comments, No. 1057, Network Information Center, SRI International, June 1988.

[15] B. Liskov, Distributed programming in ARGUS, Comm. ACM, 31(3) (March 1988) 300–312.

[16] K.P. Birman and T.A. Joseph, Reliable communication in the presence of failures, ACM Trans. Computer Systems, 5(1) (February 1987) 47–76.

[17] K.P. Birman, A. Schiper and P. Stephenson, Lightweight causal and atomic group multicast, ACM Trans. Computer Systems, 9(3) (August 1991) 272–314.

[18] K. Lin and J.D. Gannon, Atomic remote procedure call, IEEE Trans. Software Engineering, 11(10) (October 1985) 1126–1135.

[19] W. Zhou, The SRPC (Simple Remote Procedure Call System) Reference Manual, Department of Information Systems and Computer Science, National University of Singapore, 1992.

[20] A. Sinha, Client-server computing, Comm. ACM, 35(7) (July 1992) 77–98.

[21] J. Gray and A. Reuter, Transaction Processing, Morgan Kaufmann, San Mateo, CA, 1993.

[22] M. Kong, T.H. Dineen, P.J. Leach, E.A. Martin, N.W. Mishkin, J.N. Pato and G.L. Wyant, Network Computing System Reference Manua., Prentice-Hall, Englewoods Cliffs, NJ, 1990.

[23] B.N. Jain and A.K. Agrawala, Open Systems Interconnection: Its Architecture and Protocols, Elsevier, Amsterdam, 1990.

[24] J.M. Purtilo and P. Jalote, A system for supporting multi-language versions for software fault tolerance, Proc. 19th Int. Symposium on Fault Tolerant Computing, Chicago, IL, 1989, pp. 268–274.

[25] W. Zhou, A rapid prototyping system for distributed information system applications, J. Systems and Software, 24(1) (1994) 3–29.

Wanlei Zhou received the BEng and MEng degrees from Harbin Institute of Technology, Harbin, China in 1982 and 1984, respectively, and the PhD degree from the Australian National University, Canberra, Australia in 1991. He is currently a Lecturer in the School of Computing and Mathematics, Deakin University, Geelong, Australia. Before joining Deakin University, Dr Zhou was a programmer in Apollo/HP at Massachusetts, USA, a lecturer at the National University of Singapore, Singapore, and a lecturer at Monash University, Melbourne, Australia. His research interests include distributed computing, computer networks, performance evaluation and fault-tolerant computing.