Interoperable Concurrency Support with SIR Middleware*

Eugene F. Fodor (fodor@cs.ucdavis.edu, http://avalon.cs.ucdavis.edu/)
Ronald A. Olsson (olsson@cs.ucdavis.edu)
Department of Computer Science, University of California, Davis, USA 95616

Abstract

This paper presents Synchronizing Interoperable Resources (SIR). SIR extends to multi-program environments the concurrent communication mechanisms in the SR concurrent programming language. This paper discusses design and implementation issues including implicit binding, a mechanism for providing seamless concurrent communication. It also examines some performance results of SIR as well as presenting qualitative analysis.
Keywords: Interoperable, Concurrent Programming, Distributed Programming

1. Introduction

Many development environments (such as the Java Development Toolkit) now support some notion of concurrent programming, yet few mechanisms exists for concurrent communication support between programs [10]. Furthermore, such concurrency support mechanisms should aim to make communication between remote programs transparent, a quality known as *location transparency*. These concurrent mechanisms should also be able to address *interoperability* aspects by transcending language boundaries. For example, as part of a federation database, a database program written in Java might transparently invoke and control operations or methods physically located in a distributed database program written in Ada. In addition to databases, other important related distributed applications include scientific applications, web applications, and operating system services such as file servers.

This paper presents an approach to the problem of *inter-program* concurrency. It presents SIR: Synchronizing Interoperable Resources. SIR extends to multi-program environments the concurrent communication mechanisms in the SR concurrent programming language [1], namely remote procedure call (RPC), asynchronous message passing, rendezvous, and dynamic process creation. The SIR extensions to the SR programming model and implementation are minor. SIR introduces the notion of *implicit binding*, a concept for doing seamless inter-program concurrency. SIR's expressive communication allows the programmer to work at a high conceptual level, without getting bogged down in the many details of other approaches.

Our work is intended to illustrate the general need for more flexible and easier to use communication among multiple concurrent programs. SIR represents a step in that direction. Our hope is that SIR will influence the further evolution of interoperable technologies like CORBA.

The rest of this paper is organized as follows. Section 2 discusses related work. Section 3 explains relevant key aspects of the SR programming language. Section 4 describes the design of SIR. Section 5 presents a distributed file server and client written in SIR. Section 6 contains implementation issues for the SIR compiler and run-time system. Section 7 shows some promising preliminary performance comparisons with both other approaches and SR. Section 8 discusses additional design issues. Finally, Section 9 concludes the paper.

2. Related Work

Many systems contain mechanisms for performing distributed communication. They typically provide some kind of Remote Procedure Call (RPC) [2]. For example, DCOM[3], Java RMI[8], ILU[5], and CORBA[4] are forms of object request brokers that allow remote methods to be invoked through remote objects. Converse[6] provides interoperability between single, parallel applications via runtime support.

Unfortunately, these (and other) existing mechanisms that address inter-program communication are complicated, so system designers continue to seek novel, simpler methods to express such communication. The variety of mechanisms that SIR provides allows the programmer to more intuitively design and implement the desired application. Although SIR does not provide the full functionality of, say, CORBA, it does provide more flexible, seamless inter-program concurrent communication.

*This work is partially supported by Z–World, Inc. under the UC MICRO Program and by the United States National Security Agency University Research Program.

3. SR Background

SR's communication mechanisms help the programmer express useful communication patterns with relative ease [1]. *Operations*, declared with the keyword **op**, provide the fundamental abstraction for invocation and service. For example, Figure 1 shows parts of an SR file server. This server services **fopen** invocations by first creating a process to run the **proc fopen**. The **fopen** invocation then returns a file descriptor structure (record) that contains *capabilities* to reference (i.e., point to) local operations for the corresponding file. After replying with this result to unblock its caller, the process servicing the **fopen** invocation continues execution concurrently with its invoker. It provides service for its local operations (**rd**, **wr**, and **cl**) through the **in** statement, which provides rendezvous-style servicing. Figure 2 shows the corresponding client resource, which invokes the local operations via the capabilities of the file descriptor returned by **fopen**. Such an SR program would also contain a main resource that runs first. Usually, the main resource creates and distributes other resources to other hosts. For this program, the main resource creates an instance of the server resource FileServer and an instance of the client resource FileCat; it passes the FileCat instance a capability to the operation **fopen** of the FileServer instance. A noteworthy limitation of standard SR is that resource FileCat and resource FileServer must reside in the same program.

```
resource FileServer
  import Defs # include type definitions
  op fopen(user, fname: name; m: accessmode)
    returns fd: file_descriptor
body FileServer()
  proc fopen(user, fname, m) returns fd
    # operations local to fopen
    op rd: fread, wr: fwrite, cl: fclose
    var f := open("dfs/" || user || "/" || fname, m)

    fd := file_descriptor(rd, wr, cl)
    # return result via 'reply' to un-
block invoker;
    # this process and in-
voker now run concurrently
    reply
    # rendezvous-style servicing for lo-
cal operations
    do true ->
      in rd(ln) returns status ->
        status := read(f, ln)
      [] wr(ln) -> write(f, ln)
      [] cl() -> close(f)
         exit #break out of loop
      ni
    od
  end
end
```

Figure 1. SR FileServer resource provides operation fopen to other client resources.

4. Design of SIR

SIR aims to provide interoperability by using the full concurrent communication mechanisms of SR. Therefore, only minimal changes were to be made to these mechanisms. For instance, SIR retains SR's mechanisms for invoking and servicing operations so that processes in SIR programs can communicate with one another via rendezvous, asynchronous message passing, or RPC expressed using the same syntax and semantics as in SR. Unlike in SR, however, SIR invocations may cross program boundaries.

At a high level, our model consists of communication between three programs: the remote service requester (RSR), the remote service host (RSH), and the registry program (RP). The RSR is the client, the RSH is the server, and the RP negotiates the connection between the other two. More details appear in Section 6.

To enable inter-program communication among SR programs, we added registration (making operations available to other programs) and bind (connecting to another program's registered operations) mechanisms. Specifically, we added the following built-in functions:

register(regname, oper) Registers the operation **oper** with the RP on the local host under the name **regname**.

unregister(regname) Removes the registry entry from the RP.

bind(regname, host, opcap) Contacts the RP on the specified host. Assigns the remote operation to the capability (i.e., pointer to operation) **opcap**.

unbind(opcap) Dis-associates **opcap** from the remote operation.

These functions define how to accomplish *explicit binding*, the explicit assignment of an operation to a capability between two different programs.

Implicit binding serves as a complement to *explicit binding* by allowing remote operations to be assigned to capabilities without explicitly calling bind or register. Parameter passing and return values provides the mechanism for such implicit assignments.

Figure 3 shows the time-line for an example of explicit and implicit binding. A server program (an RSH) first registers its operation X by passing a capability for X to the RP. The RP returns a boolean value back to the server program to confirm the registration. The client program (an RSR) then *explicitly* binds to operation X by contacting the registry program, which copies the capability in the appropriate registry entry to C1. The client program invokes operation X on the remote server program via the capability

```
# creator passes FileCat capability for fopen
resource FileCat ( fopen: cap (name; name; mode)
    returns file_descriptor; fname:name)
  import Defs # include type definitions
  external getlogin() returns name
  var fd: file_descriptor;  var ln: line

  fd := fopen(getlogin(), fname, READ)
  do fd.r(ln) != EOF -> write(ln) od
  fd.c()

end
```

Figure 2. SR FileCat resource prints out the contents of desired file.

C1. X returns a capability for another operation Y, which also is located within the server. This capability is assigned to C2, and the client program may then invoke Y via C2. C2 thereby *implicitly* binds to Y without an explicit call to bind. Note how at least one explicit bind needs to be made before programs can communicate via implicit binding.

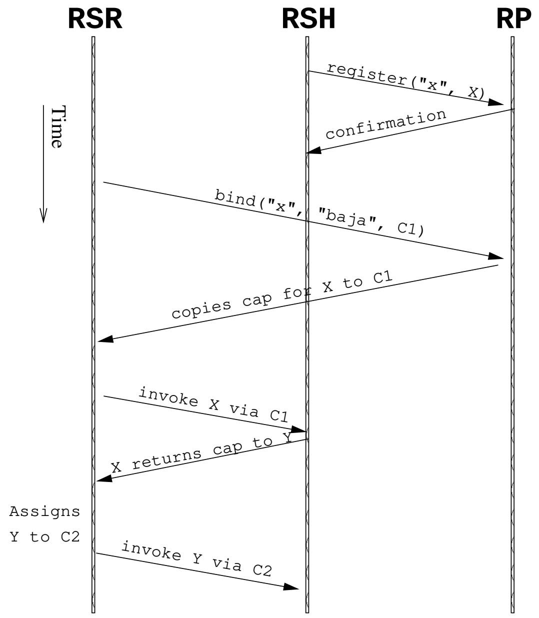

Figure 3. Timeline of explicit and implicit binding interactions. Arrows pointing to the right indicate invocations; arrows pointing to the left represent return or result values.

Type-checking for explicit binding must be performed at run time via SIR run-time system support. This requirement results from the fact that the parameters **opcap** and **oper** of the functions **bind** and **register**, respectively, may have any signature and are bound at run time. On the other hand, type-checking for implicit binding can be performed at compile-time via preexisting SR compiler actions.

5. Example: SIR Distributed File System (DFS)

Figures 4 and 5 present selected parts of two interacting SIR programs: a SIR distributed file server (DFSServer) and a representative client program (DFSCat).[1] The server's main resource distributes the server program to multiple hosts specified in its configuration. Each host in the DFS provides a directory per user and explicitly exports (i.e., registers) operations including create file, delete file, open file, and list directory. A process in each host waits for such remote invocations.

Figures 4 and 5 contain examples of explicit and implicit binding between the server program and a client program. The server program explicitly registers the operation **fopen** at line 6. The client program correspondingly performs an explicit bind at line 13. Aside from the explicit bind and register, the client and server have not changed significantly from the single program presented in Figures 1 and 2. The primary difference is how the client obtains a capability from the server. In Figures 1 and 2, the capability is passed as a parameter to the resource client. In Figure 5, the capability cannot be directly passed to the client resource since it is in a separate program. Explicit binding is only needed to open the file; all other remote operations are bound implicitly from the passed file descriptor. Capabilities **w**, **r**, and **c** of the file descriptor **fd** of the DFSCat program implicitly bind to the local operations **wr**, **rd**, and **cl** of the remote **fopen** invocation, respectively.

6. Implementation of SIR

We have built a working version of SIR's compiler and runtime system. The implementation, which extends the standard SR compiler and run-time system (RTS), required additional code to support the new built-in functions and to add *network identifiers* (see below) to data structures for capabilities. Implementing these changes required approximately 340 lines of new code in the RTS and 263 additional lines of code in the compiler. Also, the registry program (RP) was added to the RTS. The implementation of the registry program totals about 669 lines of code. The initial implementation took approximately 1 programmer month.

Invocation of operations via capabilities is the same regardless of whether the capability is bound explicitly or implicitly. Upon invocation, an invocation block is generated at the client side and is sent over the connection to the RSH.

[1]Type definitions, initialization code, and other SR code have been omitted for brevity.

```
resource DFSServer()
  import Defs #include type definitions
  op fopen(user, fname: name; m: accessmode)
    returns fd: file_descriptor
  # explicitly register fopen
  register("fopen", fopen)

  proc fopen(user, fname, m) returns fd
    op rd: fread, wr: fwrite, cl: fclose  # local operations
     var f := open("dfs/" || user || "/" || fname, m)

    fd := file_descriptor(rd, wr, cl)
    # reply implicitly returns capabilities for rd, wr, cl
    reply  # return result to invoker
    do true ->
      in rd(ln) returns status ->
        status := read(f, ln)
      [] wr(ln) -> write(f, ln)
      [] cl() -> close(f)
        exit #break out of loop
      ni
    od
  end
end
```

Figure 4. SIR server program: DFSServer resource

These invocations rely on knowing the *network identifier* of the desired operation. The network identifier contains the TCP host and port on which the RSH is listening; it is stored as part of each operation's "value", so it can be copied when the operation is assigned to a capability.

6.1. Explicit Binding

The register function starts makes an operation available for remote invocation by sending an operation's capability to the RP. This capability is hashed into a table of registry entries based on the registration name.

The bind function provides a virtual connection between the RSR and RSH. The RP establishes this connection by forwarding the capability of the desired RSR operation to the RSH. This capability creates the virtual connection to the RSH. When used, the capability activates an actual network connection using the referred operation's network identifier. A hash table allows for the caching of multiple simultaneous network connections. Unbind simply disconnects and cleans up any data structures as necessary.

For register and bind, the compiler-generated code encodes the signatures of capabilities and operations into RTS calls to provide dynamic type checking. When an operation is registered, this encoding is sent as part of the registry entry to the RP. When an operation is bound, an RSR initiates the RTS comparison of the encoded signature. Once bound, an operation can be invoked without requiring further type checking. The encoding is a simple text representation of the signature.

```
resource DFSCat()
  import Defs
  external getlogin() returns name
  var fopen: cap (name; name; accessmode)
                 returns file_descriptor
  var machine, file: name
  var arg:name

  procedure cmd_cat(fname: name; server: string[*])
    var fd:file_descriptor
    var ln: line
    # explicitly bind to operation fopen
    bind("fopen", server, fopen)
    # implicitly bind to operations rd and cl
    fd := fopen(getlogin(), fname, READ)
    do fd.r(ln) != EOF -> write(ln) od
    fd.c()
    unbind(fopen)
  end

  getarg(1, arg); parse_filename(arg, machine, file)
  cmd_cat(file, machine)
end
```

Figure 5. SIR client program: DFSCat resource

6.2. Implicit Binding

Early versions of SIR did not support implicit binding because they relied on a static mapping of remote program descriptors (integer values) to physical connections. By using network identifiers, capabilities make implicit binding possible since capabilities store the location of the original operation. Furthermore, network identifiers allow capabilities that refer to remote program operations to be distributed within a single program or even passed on to other programs.

Type-checking for implicit binding did not require any changes to the SR compiler or RTS.

7. Performance

We have obtained preliminary SIR performance results. We compared SIR with two freely available implementations of the CORBA 2.x specification: Mico[9] and OmniORB2[7]. The former serves primarily as an educational 2.2 compliant ORB; the latter serves as a high-performance 2.0 compliant ORB. A second test compares the performance of SIR invocations to that of standard SR invocations.

Both set of tests were run on an isolated 10 Mbps Ethernet LAN with 3 machines. The machines used in the experiment were a Dual SMP 233 MHz Pentium, a 120 MHz Pentium laptop, and a 120 MHz Pentium PC.

The CORBA programs were written in C++. The test application for comparison with CORBA is a simple time service application that uses RPC. Each test client runs for 1 minute counting the number of remote invocations of the

	Configuration 1	Configuration 2	Configuration 3			Lines of code
			120MHz	120MHz	233MHz SMP	
Mico	560	366	198	203	400	88
SIR	1531	595	443	438	891	33
OmniORB2	2110	928	711	725	856	84

Table 1. Invocations per second of OmniORB, Mico and SIR. Configurations 1, 2, and 3 consist of a single client and single server running on one machine, a single client and single server running on different machines, and 3 clients and a single server running on 3 machines, respectively.

	SR	SIR	Invocations/Round
DFS List	250	294	1
DFS Copy	44	54	6

Table 2. Rounds per second of SIR versus SR. DFS List is a single remote operation. DFS Copy is an aggregation of different remote operations.

time service. Table 1 shows the average number of invocations per second for five test runs under three different configurations. The last column shows the total lines of code required. As shown, the SIR code requires significantly less effort due to the details that are required to setup communication in CORBA.

In Table 2, the second test set compares SIR against SR by averaging number of *rounds* per second using code similar to that presented in Section 5. Each round consists of a number of invocations that comprise the desired action (e.g., directory listing, file copy). The SR program provides a basis for comparison by running a file server and client on different hosts within the same program. In this way, this test is designed to measure the overhead between separate SIR programs and a single SR program. The first row shows a fine grain comparison by looking at the performance of a single DFS operation that lists the files for the current user. The second row shows a larger grain comparison by looking at the performance of DFS copy, a client program for copying files between DFS hosts. As the table shows, the performance difference between SR and SIR is not significant. SIR outperforms SR due to optimizations added to the invocation handling code.

These performance results are intended to give a general idea of SIR's performance versus the performance of other approaches. However, direct comparisons are not necessarily appropriate (without further work) because CORBA ORB's and SIR are different in important aspects. For example, ORB's support the notion of remote objects, but SIR does not. On the other hand, SIR supports more communications paradigms. CORBA also provides a much richer set of naming and other services, while SIR is relatively simple.

8. Discussion

As illustrated by the example in Section 5, implicit binding provides a seamless way to add inter-program concurrency support. Unfortunately, implicit binding complicates other aspects of distributed computation. The standard SR run-time system contains mechanisms for termination and deadlock detection. These mechanisms become more complicated in a multi-program environment. In standard SR, the RTS detects program quiescence and decides to terminate the program by keeping track of invocation requests through a special monitoring process called **srx**. However, performing accounting on invocation requests between SIR programs complicates matters since an additional mechanism is required to monitor pending invocations between programs. Furthermore, the semantics for quiescence change. A program with registered operations that would be quiescent in traditional SR may have another program bind and use those registered operations. Alternatively, a program might be waiting on an operation that it has implicitly passed to another program. In this scenario, even if all explicitly bound operations (obviously, these are easier to track) have been resolved (unbound and unregistered), we should not allow the program to terminate until all external processes using the implicitly bound operation are finished with it. Since the conditions for termination may be only partially satisfied, a new, more global view is required to detect such conditions. A possible solution to this problem would be to create a monitor process to help manage termination. Currently, SIR relies mostly on the programmer to detect deadlock and termination conditions.

Another shortcoming of the current SIR implementation is the lack of exception handling in the event that a bind operation fails. The loss of scope or the destruction of a remote operation further complicates this problem since capabilities can become dangerous dangling references. SIR also currently does not have remote resource (i.e., object) support, and we hope to address this issue in future work as well.

The current design of SIR has two inherent limitations: operations may only be registered on the local machine, and the registration name must be unique for each operation. These problems can be solved, respectively, by adding another parameter to the register function and by using path names as Java RMI does. The former limitation also occurs in approaches like Java RMI; security issues motivate this limitation[8].

Currently, we have only investigated SIR to SIR communication, but intend to extend this work to allow other languages to utilize SIR operations. For instance, we are investigating ways to map concurrent communication mechanisms between programming languages. We are considering several approaches. One approach would use SIR as a definition language to explicitly define mappings. Another would simulate mechanisms through generated stub code and run-time system calls.

9. Conclusion

We have presented SIR, a flexible development system for interoperable, concurrent applications. Since SIR naturally extends the SR language mechanisms, it helps make communication between programs more intuitive.

SIR's implicit binding provides an easy way of dividing single, distributed programs since major portions of the code can remain unchanged. This desirable characteristic of implicit binding should extend easily to facilitate interoperability between different, distributed applications without requiring major code changes. For example, a distributed database could communicate and share data with various distributed scientific applications without requiring the applications to be re-written. We suggest that other approaches for interoperability should consider incorporating a mechanism similar to implicit binding.

References

[1] G. Andrews and R. Olsson. *The SR Programming Language: Concurrency in Practice*. Benjamin/Cummings, Redwood City, CA, 1993.

[2] A. Birrell and B. Nelson. Implementing remote procedure calls. *ACM Trans. Computer Systems*, 2(1):39–59, Feb. 1984.

[3] N. Brown and C. Kindel. *Distributed Component Object Model Protocol (DCOM/1.0)*, Jan. 1998. Microsoft.

[4] O. M. Group. *The Common Object Request Broker: Architecture and Specification.*, Feb 1998.

[5] B. Janssen, M. Spreitzer, D. Larner, and C. Jacobi. *ILU 2.0alpha12 Reference Manual*, Nov. 1997. XEROX PARC.

[6] L. V. Kale, M. Bhandarkar, R. Brunner, and J. Yelon. Multiparadigm, Multilingual Interoperability: Experience with Converse. In *Proceedings of 2nd Workshop on Runtime Systems for Parallel Programming (RTSPP) Orlando, Florida - USA*, Lecture Notes in Computer Science, March 1998.

[7] S.-L. Lo and S. Pope. The Implementation of a High Performance ORB over Multiple Network Transports. In *Middleware 98: IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing*, 1998. http://www.uk.research.att.com/omniORB/omniORB.html.

[8] S. Microsystems. Java remote method invocation specification. July 1998.

[9] K. Römer, A. Puder, and F. Pilhofer. Mico 2.2.1. http://diamant-atm.vsb.cs.uni-frankfurt.de/~mico/, Oct. 1998.

[10] F. B. Schneider. On concurrent programming. *Communications of the ACM*, 4(41):128, Apr. 1998.