Concurrent Language Support for Interoperable Applications ⋆

Eugene F. Fodor and Ronald A. Olsson

Department of Computer Science
University of California, Davis, USA, 95616
{fodor, olsson}@cs.ucdavis.edu
http://avalon.cs.ucdavis.edu

Abstract. Current language mechanisms for concurrency are largely isolated to the domain of single programs. Furthermore, increasing interest in concurrent programming encourages further research into interoperability in multi-lingual, multi-program settings. This paper introduces our work on Synchronizing Interoperable Resources (SIR), a language approach to begin achieving such interoperability with respect to concurrent language mechanisms.

1 Introduction

Developers of distributed software systems need simpler, more efficient ways of providing communication between different, remote programs in multi-lingual environments. Figure 1 shows an example of this multi-lingual interaction problem in a federation of databases and a collection of client programs.

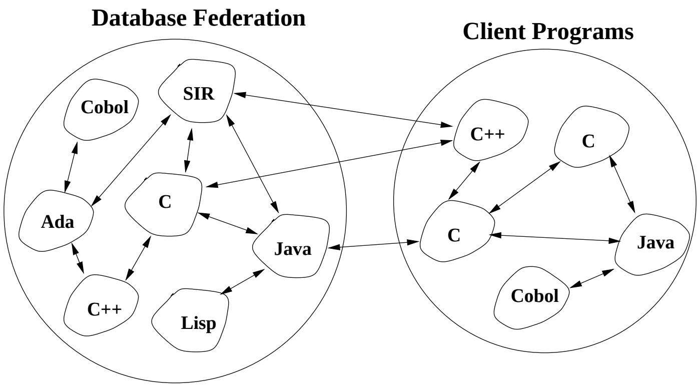

Fig. 1. Interoperability in a Database Federation

⋆ This work is partially supported by the United States National Security Agency University Research Program and by ZWorld, Inc. and the State of California under the UC MICRO Program.

Many systems contain mechanisms for performing distributed communication. They typically provide some kind of Remote Procedure Call (RPC) [1]. For example, DCOM[2], Java RMI[3], ILU[4], and CORBA[5] are forms of object request brokers that allow remote methods to be invoked through remote objects. Direct support for interoperability of concurrency control mechanisms under these systems is limited. For example, CORBA's Concurrency Control Service [6] restricts concurrent control to a transaction based two-phase locking protocol commonly used in database systems. Converse[7] provides multi-lingual interoperability for concurrent languages via run-time support within the scope of a single, parallel application.

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. This paper presents an alternative approach to this problem. Synchronizing Interoperable Resources (SIR) extends the communication mechanisms in the SR concurrent programming language [8]. SR provides concurrent communication mechanisms like RPC, semaphores, asynchronous message passing, rendezvous, and dynamic process creation. This variety of mechanisms 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 inter-program communication.

SIR is intended to benefit distributed system researchers and developers. 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. The SIR extensions to the SR programming model and implementation are minor. SIR also facilitates rapid prototype development.

The rest of this paper presents an overview of the design and implementation of SIR, including the requirements and interactions of the SIR compiler and run-time support system. It presents some preliminary performance results, which are promising. This paper also discusses trade-offs with other approaches and some of the other issues that have arisen so far in this work, which will be the basis of future work.

2 SR Background

The expressive power of SR's communication mechanisms helps the programmer create complex communication patterns with relative ease [8]. For example, a file server can service "open" invocations only when file descriptors are available. Furthermore, the server might want to schedule invocation servicing by some priority. The SR code fragment in Figure 2 accomplishes this easily through SR's **in** statement, which provides rendezvous-style servicing. Both synchronization and scheduling expressions (**st** and **by**, respectively) constrain the execution of each arm of the **in** statement. The priority (`pri`) could be mapped to an array of host names, thus allowing preference for specific hosts. In SR, the types of an operation's parameters and its return type are collectively called the operation's *signature*.

```
process fileserve()
  do true ->
    in rmt_open(fname, pri) st fdcnt > 0 by pri ->
       rfs_service(fname, OPEN)
       fdcnt--
    [] rmt_close(fname, pri) by pri ->
       rfs_service(fname, CLOSE)
       fdcnt++
    ...
    ni
  od
end
```

Fig. 2. SR file server

3 Design

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 5.

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) Disassociates **opcap** from the remote operation.

This design implies that type checking must be performed at run time because the parameters **opcap** and **oper** of the functions **bind** and **register**, respectively, may have any signature and are bound at run time. Our initial design has two 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[3].

4 Example

SIR communication naturally extends traditional SR mechanisms and allows for rapid prototyping. Consider, for example, building a server for some online database. The clients and server both create remote queues using SR's send and receive mechanisms to establish a complex dialog. Each client first registers its queue to the RP and then binds the known server queue ("kq") to its capability (**capqueue**). In turn, the server binds to the unique name transferred by the client and replies on the corresponding capability (**remqueue**) after performing some data processing. The SIR code fragments in Figures 3 and 4 show this interaction. Note that the program could actually be written more directly using RPC; however, this more complicated version better illustrates the expressive power of SIR.

```
#Each client requests a unique name
#from its local RP.
uniqname(uid)
register(uid, myqueue)
do not done->
  bind("kq", "host", capqueue)
  #myhost() returns local host name.
  send capqueue(uid, myhost(), data)
  #Receive processed data.
  receive myqueue(data)
od
unbind(capqueue)
unregister(uid)
```

Fig. 3. Code for multiple, different clients

```
register ("kq", queue)
do not done ->
  receive queue(name, host, data)
  #Invoke datahndlr as
  #separate thread.
  send datahndlr(name, host, data)
od
unregister("kq")

proc datahndlr(name, host, data)
  var remqueue:cap (datatype)
  bind(name, host, remqueue)
  #Process data (not shown).
  #Send data after processing.
  send remqueue(data)
  unbind(remqueue)
end
```

Fig. 4. Code for single server

5 Implementation

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 map identifiers for remote programs into existing RTS data structures and to support the new built-in functions. Implementing these changes required approximately 340 lines of new code in the RTS and 263 additional lines of code in the compiler. Also, a special program called the registry program (discussed below) was added to the RTS. The implementation of the registry program totals about 669 lines of code. The implementation took approximately 1 programmer month.

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.

The bind function provides a direct connection between the RSR and RSH. The RP establishes this connection by forwarding the capability of the RSH's registered operation to the RSR. The RSR then sends back an acknowledgment to the RP to complete the request and thus frees it to establish other connections. If the RSR times-out or receives an error from the RP, the call to bind returns false. Upon invocation, an invocation block is generated at the client side and is sent over a connection to the RSH. Such connections between clients and servers are created only upon invocation and are cached for later use. Unbind simply disconnects and cleans up any data structures as necessary. Figure 5 shows this interaction at a high level.

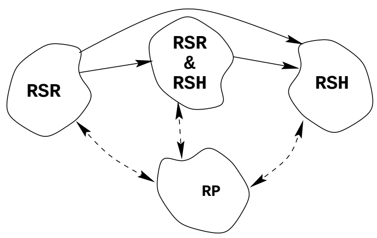

Fig. 5. Sample interactions among RSR (Remote Service Requester), RSH (Remote Service Host), and RP (Registry Program). Note that a program can act both as an RSR and an RSH. The solid lines indicate invocations; the dashed lines imply the low-level communication with the RTS for register and bind.

6 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[10]. The former serves primarily as an educational 2.2 compliant ORB; the latter serves as a high-performance 2.0 compliant ORB. The following performance comparisons with CORBA are meant to only serve as a rough guide of SIR performance.

The 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 is a simple time service application that uses RPC. Each test client runs for 1 minute counting the number of remote invocations of the 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.

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.

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.

	Config 1	Config 2	Config 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

7 Issues

The design and implementation of SIR has raised some interesting issues. In traditional SR, capabilities can be casually copied and passed within a single program. For example, in the following program fragment, process A sends process B a capability to foo. Process B assigns this capability to y and then invokes foo via y.

```
op x(cap ())

process A
  op foo()

  send f(foo)
  in foo() ->
    write("bar")
  ni
end
```

```
process B
  var y:cap ()
  receive x(y)
  y()
end
```

Interoperating SIR programs do not provide this sort of functionality *if* limited to passing capabilities via register and bind. Under this limitation, register and bind are required for inter-program communication, since they explicitly pass the context information needed for remote invocation. However, SIR escapes this problem by introducing the notion of implicit binding. Implicit binding allows a program to pass a capability to another program as done in traditional SR without requiring a call to bind. Explicit binding allows the same communication mechanisms to be used regardless of whether they are local or remote—a property known as location transparency [5]. Implicit binding enhances location transparency by reducing the need for explicit binds.

Implicit binding complicates other aspects of distributed computation. For instance, termination detection mechanisms present in traditional SR 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. Even if all explicitly bound operations (obviously, these are easier to track) are resolved, we should not allow the program to terminate until all external processes using it are blocked. 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.

Finally, 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.

8 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.

Currently, we have only investigated SIR to SIR communication, but intend to extend this work to allow other languages to utilize SIR operations. SIR also currently does not have remote resource (i.e., object) support, and we hope to address this issue in future work as well. Also, we will look further at SIR performance issues against other interoperable packages. Specifically, we are looking at the performance of binds, registers, and invocations as well as doing qualitative comparisons of communication mechanisms. We are also investigating better methods for partial, distributed termination, implicit binding, and exception handling. We are also extending SIR to include resources (objects) to provide a better basis for comparison with other object-oriented approaches.

We are also developing some practical applications with SIR. For example, we wrote a distributed, HTTP server in SR, and are currently augmenting it with SIR mechanisms. Additionally, we have also written a simple SIR distributed file server.

Acknowledgements

We thank Greg Benson for early technical discussions on this work. We thank the anonymous referees for helpful comments on the paper.

References

1. Andrew Birrell and Bruce Nelson. Implementing remote procedure calls. *ACM Trans. Computer Systems*, 2(1):39–59, February 1984.
2. Nat Brown and Charlie Kindel. *Distributed Component Object Model Protocol (DCOM/1.0)*, January 1998. Microsoft.
3. Sun Microsystems. Java remote method invocation specification. July 1998.
4. Bill Janssen, Mike Spreitzer, Dan Larner, and Chris Jacobi. *ILU 2.0alpha12 Reference Manual*, November 1997. XEROX PARC.
5. Object Management Group. *The Common Object Request Broker: Architecture and Specification.*, Feb 1998.
6. Object Management Group. *CORBAservices: Common Object Services Specification.*, July 1998.
7. L. V. Kale, Milind Bhandarkar, Robert Brunner, and Joshua 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.
8. Gregory Andrews and Ronald Olsson. *The SR Programming Language: Concurrency in Practice*. Benjamin/Cummings, Redwood City, CA, 1993.
9. Kay Römer, Arno Puder, and Frank Pilhofer. Mico 2.2.1. http://diamant-atm.vsb.cs.uni-frankfurt.de/~mico/, October 1998.
10. Olivetti & Oricale Reasearch Laboratory. Omniorb2. http://www.orl.co.uk/omniORB/, October 1998.