SIR: Inter-program Concurrency Support for SR Programs*

Eugene F. Fodor and Ronald A. Olsson
Department of Computer Science
University of California, Davis
Davis, CA, USA 95616-8562

{fodor,olsson}@cs.ucdavis.edu

Notes to Publisher:

- *Please send correspondence regarding this paper to* Olsson

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. This paper also compares SIR with CORBA and other systems that provide interoperability.
Keywords: Inter-Program Concurrent Programming, Concurrent Programming Languages, Interoperability, Concurrent Programming, Distributed Programming

*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.

1 Introduction

Many development environments (such as the Java Development Toolkit) now support some notion of concurrent programming, yet few mechanisms exist for concurrent communication support between programs. 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.

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, however, limited in the kinds of concurrency they provide and in the ways in which programs establish connections.

This paper presents a different 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 [6], 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 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. For example, the effect of implicit binding can be achieved in CORBA using object references, but it is cumbersome. (A detailed example appears in Sections 4 and 5.)

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. SIR shares some of the same goals as CORBA and other interoperable technologies. Although SIR does not provide the full functionality of, say, CORBA, it does provide a different model for inter-program concurrent communication using the native concurrency mechanisms of the language. In that respect, our hope is that SIR will influence the further evolution of interoperable technologies like CORBA in providing inter-program concurrency.

The rest of this paper is organized as follows. Section 2 explains relevant aspects of the SR programming language. Section 3 describes the design of SIR. Section 4 presents a distributed file server and client written in SIR. Section 5 presents an equivalent CORBA implementation of the distributed file server and client. Section 6 describes our implementation of the SIR compiler and run-time system. Section 7 shows some promising preliminary performance comparisons with other approaches and SR. Section 8 compares SIR and CORBA and discusses additional SIR design issues. Section 9 discusses related work and compares SIR to other efforts. Finally, Section 10 concludes the paper.

2 SR Background

Central to SR [6] are the notions of virtual machines, resources, and operations.

A *virtual machine* (VM) represents an address space. VMs are used for distributing a program onto physical machines. Each VM resides on one physical machine. VMs are created dynamically.

SR's primary modular component is the resource. Instances of resources are dynamically created. Processes execute within a particular resource instance and have (shared) access to variables and operations within that instance. Each SR program contains a *main* resource that runs first. Usually, the main resource creates and distributes instances of other resources to other hosts.

An operation can be considered a generalization of a procedure and enables processes to interact. Operations, declared with the keyword `op`, provide the fundamental abstraction for invocation and service. They help the programmer express useful communication patterns with relative ease.

Given that VMs, resources, and operations are created dynamically, SR uses *capabilities* for each. A capability acts as a pointer and can be assigned to variables and passed as parameters, thus permitting, for example, dynamic communication paths.

Below, we illustrate the salient features of SR by showing code for a simple distributed file system, which includes a file server and a client that uses the file server to output files. (This example is based heavily on a similar one in [6].)

The main resource for this program is shown in Figure 1. It and the other resources use the common type definitions shown in Figure 2. (For brevity, the code fragments omit some type definitions and initialization.) When executed, 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.

Figure 3 shows the key 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 (`rd`, `wr`, and `cl`) 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. Recall that `FileCat` obtains capabilities for these local operations, which it assigns to its local variable `fd`; it later invokes `fopen`'s local operations through `fd`'s fields.

Figure 4 shows the client resource. It first interacts with the user to get the name of the file to output. It then interacts with the file server to open, read, and close the file. Specifically, it invokes operations within the file server via the capabilities of the file descriptor returned by `fopen`.

A noteworthy limitation of standard SR is that the instances of the `FileCat`, `FileServer`, and main resources must reside in the same program for them to communicate.

3 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 invocations in SR, however, invocations in SIR may cross program boundaries.

At a high level, our model reflects the 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 named `regname` 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. The example in Section 4 illustrates each kind of binding.

4 Example: SIR Distributed File System (DFS)

Figures 5 and 6 present a SIR version of the file system example given in Section 2. Much of the code is identical, but the major difference is that the file system now consists of two interacting SIR programs: the file server program (`DFSServer`) and the client program (`DFSCat`). (No separate main program is needed.) Whereas the connections between client and server were previously via capabilities passed as parameters to resources, the connections are now via capabilities that have been registered and explicitly or implicitly bound.

The server, shown in Figure 5, is nearly identical to `FileServer`, shown in Figure 3. The difference is that `DFSServer` registers its operation `fopen`. (Note that operation `fopen` could also now be declared to be local to `DFSServer` since its name is no longer needed outside `DFSServer`.)

The client, shown in Figure 6, is nearly identical to `FileCat`, shown in Figure 4. The difference is that `DFSCat` obtains its capability for `fopen` via explicit binding, rather than as a parameter. (It also unbinds `fopen` when finished.) In Figure 6, the capability cannot be directly passed to the client resource since it is in a separate program.

Figures 5 and 6 contain examples of explicit and implicit binding between the server program and a client program. The server program (Figure 5) explicitly registers the operation `fopen`. The client program (Figure 6) correspondingly performs an explicit bind. 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.

To summarize the two kinds of binding in the above example: The `fopen` capability in `DFSCat` is *explicitly* bound to the `fopen` operation in `DFSServer`, but the fields of `fd` are *implicitly* bound to the operations local to the `fopen` proc in `DFSServer` (because no explicit call to bind or register is needed). Note how at least one explicit bind needs to be made before programs can communicate via implicit binding.

Figure 7 shows the time-line for the execution of the code in Figures 5 and 6 in terms of interactions between the server (RSH, `DFSServer`), client (RSR, `DFSCat`), and registry programs (RP).

5 Example: Equivalent CORBA DFS

Each CORBA program may be written in any language that the ORB supports. The CORBA specification contains mappings for many languages including C++, C, Java, Smalltalk, and Ada. Other languages such as Eiffel have been added, but they are not part of the official CORBA specification (although they may be included later).

As a basis for comparison with SIR, Figures 8, 9, 10, and 11 show functionally equivalent CORBA versions of the programs presented in the previous section. The CORBA code presented here is written in C++ for the MICO ORB [7] and may differ in syntax from code written for other ORBs. Code generated from the IDL compiler, such as for the interface in Figure 8, must be compiled into both the client and the server.

CORBA differs from SIR in that it does not provide capabilities for methods directly, but rather through object references. The code in Figure 9 defines a class that implements the `File_Descriptor` interface from Figure 8. In the code in Figure 10, the object reference provided by the server is registered with the CORBA MICO name service through the call to bind. Figure 11 shows the client obtaining an object reference to the file descriptor object on the server side through the call to resolve. Note the similarity between CORBA MICO's `bind` and `resolve` and SIR's `register` and `bind`, respectively.*

*Java RMI use `rebind` and `lookup` for similar operations. ORBIX [8], another CORBA ORB, uses `registerObject` and `resolveName`, although at one time used `_bind` for the latter (now deprecated). SIR's use of the term "bind" reflects our view of what it means to bind an operation capability to an operation, i.e., the binding assigns a value to the operation capability.

A type of implicit binding may also be accomplished in CORBA by passing object references between functions. SIR like CORBA requires that only the type information for each possible reference be known at compile time so that different implementations may service the same reference. In contrast to the SIR DFS example, the CORBA example shown does not contain any implicit binding since the `File_Descriptor` object reference is passed explicitly.† Implicit binding can be provided in CORBA and is useful in implementing an indirect connection between servers as shown in Figures 12 and 13. In this simple example, the `File_Descriptor` object has been extended to include the ability to access other known file servers, so that method invocations are passed to the server on which the file is located. Binding to the `File_Descriptor` references is done implicitly through the `addserver` method. Thus, references to the `islocal` method, such as those within `open`, are now possible no resolve or bind for `islocal`. Figure 12 shows that the IDL has also changed to reflect the new operations. Section 8 further discusses general differences between SIR and CORBA.

6 Implementation of SIR

We have built a working version of SIR's compiler and run-time 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 about one 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. To generate such an invocation, the client must know the 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" and is copied when the operation is assigned to a capability. Thus, it is used when generating an invocation of the operation.

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 the RSH uses the capability, it activates an actual network connection for operation's network identifier

†The naming service itself might be considered implicit binding, since it is implemented in CORBA.

(which, recall, is part of the capability's value). A hash table allows for the caching of multiple simultaneous network connections. Unbind simply disconnects and cleans up any data structures as necessary. Figure 14 shows this interaction at a high level.

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.

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 [7] and OmniORB2 [9]. 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. The comparisons with CORBA are meant to serve only as a rough guide of SIR performance.

Both set of tests were run on an isolated 10 Mbps Ethernet LAN with three machines: 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 one 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.

In Table 2, the second test set compares SIR against SR by averaging the number of *rounds* per second using code similar to that presented in Section 4. Each round consists of a number of invocations that comprise the desired action (e.g., directory listing or 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

	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 three clients and a single server running on three 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.

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, `DFS List`, 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. One relevant comparison would be to compare a CORBA IDL for SIR (in which execution of inter-program invocations would use CORBA mechanisms) with other languages and synchronization mechanisms. Section 8 contains further comparisons between CORBA and SIR.

8 Discussion

SIR differs from CORBA in four key ways. First, capabilities in SIR allow for different patterns of communication not available in CORBA. For example, Ada supports rendezvous, but CORBA contains no support for mapping the Ada rendezvous. Second, SIR lacks support for object references. Such support could be accomplished by extending SIR to include SR resource capabilities (see Section 2). Third, SIR does not support a language independent interface specification equivalent to CORBA's IDL code. However, SR's *abstract* resources, which consist of only operation signatures, could serve the role of defining interfaces within an IDL, although they are not really language independent. (In SR and SIR, the types of an operation's parameters and its return type are collectively called the operation's *signature.*) Fourth, SIR must perform type-checking for explicit binding at run time via SIR's 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, SIR performs type-checking for implicit binding at compile time via preexisting SR compiler actions. In contrast, CORBA is able to perform the equivalent type-checking at compile time because the type information is available to it via the interface (e.g., Figure 8); as mentioned above, SIR has no equivalent of an interface.

As illustrated by the example in Section 4, implicit binding provides a seamless way to add inter-program concurrency support. Explicit binding allows the same communication mechanisms to be used regardless of whether they are local or remote—i.e., the location transparency property [5]. Implicit binding enhances location transparency by reducing the need for explicit binds.

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. Programs written using CORBA typically handle such situations via timeouts; e.g., if the name service detects that no activity occurs within a certain interval, then it can terminate itself.

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 [3].

Since our focus was primarily on providing more flexible communication among multiple concurrent programs, we have not included all features that we would like in our initial SIR implementation. In particular, SIR currently does not have any special support for SR resources or SR virtual machines (described in Section 2). Given that resources and VMs are created dynamically, SR uses capabilities for each. A natural extension to SIR would be to allow the registration and bind functions to work on capabilities for resources and VMs. Such extensions would bring SIR in line for comparison and incorporation into object-oriented standards like CORBA.

We have thus far investigated only SIR to SIR communication, but we 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, e.g., how to map Ada rendezvous to SR rendezvous. 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. Another approach would be to look at supporting language specific native communication mechanisms in technologies like CORBA.

9 Related Work

Many systems contain mechanisms for performing distributed inter-program communication. This section briefly describes some of the key systems and compares SIR with those systems.

Section 8 discussed specific differences between SIR and CORBA. At a conceptual level, the other main interoperable technologies, DCOM and Java RMI, provide mechanisms similar to those in CORBA [10], so a comparison of SIR to either of them is similar to that for SIR compared with CORBA.

One remaining issue in CORBA related to concurrency deals with the types of invocations. The CORBA 2.2 specification provides two types of synchronous invocations: synchronous and deferred synchronous. Synchronous calls simply block until the invocation is completed, whereas deferred synchronous calls (available only through the Dynamic Invocation Interface) allow results to be polled. The proposed CORBA 3.0 specification introduces CORBA Messaging to provide support for asynchronous invocations in the form of callback and polling objects. These new extensions have been defined largely in response to semantic ambiguity of the CORBA 2.2 asynchronous construct `oneway`. Although these additions enhance the specification of synchronization for basic RPC-like invocations, they lack adequate semantic mappings for the native synchronization mechanisms of various concurrent languages. For example, no direct CORBA mapping exists for Ada's `entry` construct; instead, this construct must be placed within a functional or procedural proxy (or wrapper). Concurrency services provided by the CORBA standard are also limited. For example, CORBA's Concurrency Control Service [11] restricts concurrent control to a transaction-based, two-phase locking protocol commonly used in database systems. CORBA does not provide any means for providing synchronization constraints. However, a proposed CORBA

IDL extension [12] incorporates synchronization constraint annotations. We believe that SIR-like communication can be provided in CORBA with a small change to the specification. For example, object method references could be introduced for all methods; this change would not require modification to the IDL and would preserve the abstractions that SIR provides. A more conservative approach would be to indicate explicitly (via a new keyword) which methods are allowed to act as SIR operations. The question of semantic interoperability of different concurrent languages is more difficult in practice (e.g., how to map Ada rendezvous semantics to SR rendezvous semantics, as mentioned earlier), and is beyond the scope of this paper.

Inter-language unification (ILU) done at Xerox Parc has similar goals to that of CORBA. It differs in the underlying specification language (ISL) and the languages it supports. ILU strives to be compliant with other open standards such as CORBA and so supports translation of CORBA IDL to the native ISL with some caveats. For example, some of the languages supported by ILU, such as Common LISP and Modula-3, are not part of the CORBA standard.

Concert presents a way of providing interoperability of programming languages through a preprocessor, which translates annotations into the underlying language that it extends [13, 14]. Concert/C extends C; versions of Concert for other languages (e.g., FORTRAN) were also planned. Function pointers may be passed in Concert so that the binding may look implicit. Implicit function pointers are done through the notion of a mini-stub, which pretends to be the legacy function, but really serves to start the RPC. It is interesting to note that protocol independence (ideally based on efficiency) may exist between function pointer bindings. For example, suppose the program executes on three systems — A, B, and C — and a pointer to function f() was passed from A to B, and then from B to C. Then, 'B calls f() on A' may use one protocol and 'C calls f() on A' may use another. Concert also provides heterogeneity with respect to RPC protocol selection, so that Concert can interoperate with systems such as Sun ONC RPC and DCE RPC. Concert/C also allows calls to be to be forwarded. Concert also uses a compiler generated IDL, called the Concert Signature Representation (CSR) [15], which relies on a contract mechanism to implement stubs for marshalling and unmarshalling.

Initially, Converse [16] provided interoperability between single, parallel applications via run-time support. More recent work on Converse includes general support for the monitoring, steering, and debugging of data parallel applications through the internet [17]. Similarly, the University of Oregon's Breezy for pC++ provides an architecture for monitoring and steering of data parallel applications [18].

10 Conclusion

We have presented SIR, a flexible development system for inter-program, 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, similar to the example in Section 4, a distributed database could communicate and share data with various distributed scientific applications without requiring the applications to be re-written.

We also compared SIR with similar systems. In particular, we contrasted SIR with CORBA and discussed various tradeoffs between them. SIR does not provide the full functionality of CORBA. However, SIR's model of inter-program concurrent communication uses the native concurrency mechanisms of SR. A similar approach might be suitable in other systems and is worth further exploration.

Acknowledgements

We thank Greg Benson for technical discussions on this work. We thank the anonymous referees for their 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] Gregory Andrews and Ronald Olsson. *The SR Programming Language: Concurrency in Practice*. Benjamin/Cummings, Redwood City, CA, 1993.

[7] Kay Römer, Arno Puder, and Frank Pilhofer. MICO 2.2.1. `http://diamant-atm.vsb.cs.uni-frankfurt.de/~mico/`, October 1998.

[8] IONA Technologies. *ORBIX Manuals*, 2000. `http://www.iona.com/docs/manuals/orbix/33/html`.

[9] Sai-Lai Lo and Steve 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.

[10] Gopalan Suresh Raj. A detailed comparison of CORBA, DCOM and Java/RMI. `http://www.execpc.com/~gopalan/misc/compare.html`, 2000.

[11] Object Management Group. *CORBAservices: Common Object Services Specification*, July 1998.

[12] Gregor Henze, Thomas Koch, and Bernd Kramer. Annotations for synchronization constraints in CORBA IDL. In *Third International Workshop on Services in Distributed and Networked Environments*, pages 122–8, June 1996.

[13] J. S. Auerbach, A. S. Gopal, M. T. Kennedy, and J. R. Russell. Concert/C: Supporting distributed programming with language extensions and a portable multiprotocol runtime. In *International Conference on Distributed Computing Systems*, pages 152–9, 1994.

[14] J. S. Auerbach, A. P. Goldberg, G. S. Goldszmidt, A. S. Gopal, M. T. Kennedy, J. R. Rao, and J. R. Russell. Concert/C: A language for distributed programming. In *Winter USENIX Conference*, 1994.

[15] Joshua S. Auerbach and James R. Russell. The Concert signature representation: IDL as intermediate language. In *ACM SIGPLAN Notices, IDL Workshop*, August 1994.

[16] 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.

[17] Parthasarathy Ramachandran and Laxmikant V. Kale. Web-based interaction and monitoring for parallel programs (via Conspector). In *http://charm.cs.uiuc.edu/papers/ConspectorSC99.html*, 1999.

[18] D. Brown, A. Malony, and B. Mohr. Language-based parallel program interaction: the Breezy approach. In *Proceedings of the International Conference on High Performance Computing (HiPC'95), India*, December 1995.

```
# creates and distributes other resources
resource main()
  import Defs # include type definitions
  import FileServer, FileCat

  # create an instance of FileServer
  var fs: cap FileServer
  fs := create FileServer()

  # create an instance of FileCat;
  # pass to it the FileServer's fopen
  create FileCat(fs.fopen)

end
```

Figure 1: SR main resource for file system program.

```
global Defs   # common definitions
  ...
  type name = string[20]
  type line = string[100]
  # an optype definition declares a parameterization (i.e., signature)
  # to be used in later op and cap declarations.
  optype fread(res ln: line) returns status: int
  optype fwrite(ln: line)
  optype fclose()
  type file_descriptor = rec( r: cap fread; w: cap fwrite; c: cap fclose)
  type accessmode = enum(READ, WRITE, READWRITE)
  ...
end Defs
```

Figure 2: Type definitions for file system program.

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

    fd := file_descriptor(rd, wr, cl)
    # return result via 'reply' to unblock invoker;
    # this process and invoker now run concurrently
    reply
    # rendezvous-style servicing for local 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 3: SR `FileServer` resource provides operation `fopen` to client resources.

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

  # interact with user to get file name (fname)
     ...

  # interact with FileServer to open, read, then close file
  fd := fopen(fname, READ)
  do fd.r(ln) != EOF -> write(ln) od
  fd.c()

end
```

Figure 4: SR `FileCat` (client) resource prints out the contents of desired file.

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

  # explicitly register fopen
  register("fopen", fopen)

  proc fopen(fname, m) returns fd
    ## same code as in SR FileServer (not repeated here)
  end
end
```

Figure 5: SIR server program: `DFSServer` resource.

```
resource DFSCat()
  import Defs
  var fopen: cap (name; accessmode) returns file_descriptor

  # explicitly bind to operation fopen on machine "baja"
  bind("fopen", "baja", fopen)

  ## code up to unbind is identical to FileCat code,
  ## but it's repeated here to help illustrate implicit binding

  # interact with user to get file name (fname)
  ...

  # interact with FileServer to open, read, then close file

  # implicitly bind to operations rd and cl in server
  fd := fopen(fname, READ)
  do fd.r(ln) != EOF -> write(ln) od
  fd.c()

  unbind(fopen)

end
```

Figure 6: SIR client program: `DFSCat` resource.

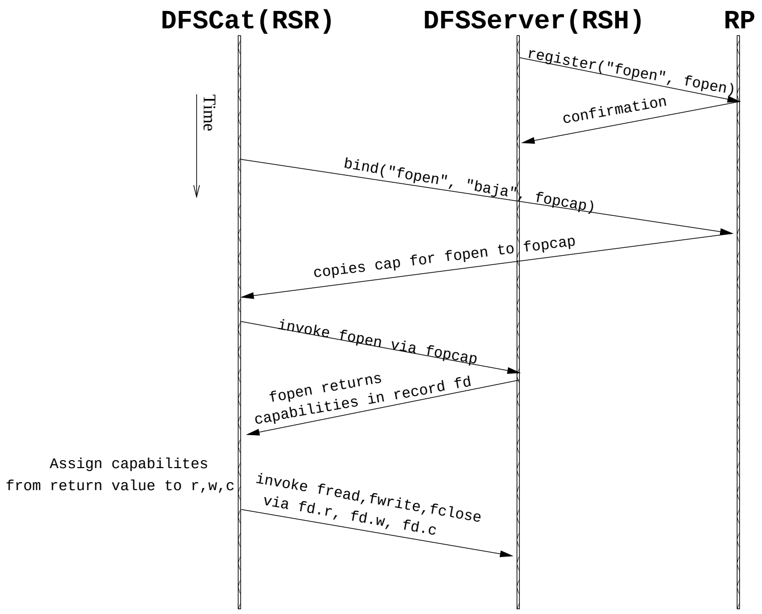

Figure 7: Timeline of explicit and implicit binding interactions. Arrows pointing to the right indicate invocations; arrows pointing to the left represent return or result values.

```
interface File_Descriptor {
   void open(in string fname);
   boolean read(out string s);
   void write(in string s);
   void close();
};
```

Figure 8: CORBA (MICO) IDL code for a file descriptor object.

```
#define MICO_CONF_IMR
#include <CORBA-SMALL.h>
#include <iostream.h>
#include <fstream.h>
#include <unistd.h>
#include <mico/CosNaming.h>
#include "dfs.h"
#include <iostream.h>
#include <fstream.h>

class File_Descriptor_impl : virtual public File_Descriptor_skel {
private:
   char fname[512];
   fstream fio;
public:

   File_Descriptor_impl () {
      fname[0] = '\0';
   }

   void open(const char *fname)  {
      fio.open(fname, ios::in|ios::out);
   }

   CORBA::Boolean read(char*& data) {
      char buf[512];
      fio.getline(buf, 512);
      data = CORBA::string_dup( buf );
      return ! fio.eof();
   }

   void write(const char* data) {
      fio << data;
   }

   void close() {
      fio.close();
   }

};
```

Figure 9: CORBA (MICO) version of `DFSServer` (Part 1/2).

```
int main( int argc, char *argv[] ) {
   CORBA::ORB_var orb = CORBA::ORB_init( argc, argv, "mico-local-orb" );
   CORBA::BOA_var boa = orb->BOA_init (argc, argv, "mico-local-boa");

   if (!boa->restoring()) {
      File_Descriptor_ptr fd = new File_Descriptor_impl();

      // get the name service
      CORBA::Object_var nsobj = orb->resolve_initial_references ("NameService");
      assert (! CORBA::is_nil (nsobj));

      CosNaming::NamingContext_var nc = CosNaming::NamingContext::_narrow (nsobj);

      CosNaming::Name name;
      name.length (1);
      name[0].id = CORBA::string_dup ("DFS");
      name[0].kind = CORBA::string_dup ("");
      nc->bind(name, fd);
   }
   boa->impl_is_ready (CORBA::ImplementationDef::_nil());
   orb->run ();

   return 0;
}
```

Figure 10: CORBA (MICO) version of `DFSServer` (Part 2/2).

```
#include <iostream.h>
#include <CORBA-SMALL.h>
#include "dfs.h"
#include <mico/CosNaming.h>

int main( int argc, char *argv[] ) {
   CORBA::ORB_var orb = CORBA::ORB_init( argc, argv, "mico-local-orb");

   CORBA::Object_var nobj =
      orb->resolve_initial_references ("NameService");
   assert (! CORBA::is_nil (nobj));

   CosNaming::NamingContext_var nc = CosNaming::NamingContext::_narrow (nobj);
   CosNaming::Name name;
   name.length (1);
   name[0].id = CORBA::string_dup ("DFS");
   name[0].kind = CORBA::string_dup ("");

   CORBA::Object_var obj;
   obj = nc->resolve (name);

   File_Descriptor_var dfsref = File_Descriptor::_narrow( obj ) ;

   CORBA::String_var s;

   if(argc < 2) {
      cerr << "Must supply a file name" << endl;
      exit(1);
   }
   dfsref->open(argv[1]);
   while(dfsref->read(s.out()))
      cout << s.in() << endl;
   dfsref->close();
}
```

Figure 11: CORBA (MICO) version of `DFSCat` client.

```
interface File_Descriptor {
   void open(in string fname);
   boolean read(out string s);
   void write(in string s);
   void close();
   boolean islocal(in string fname);
   void addserver(in File_Descriptor fd)
};
```

Figure 12: CORBA (MICO) IDL code for `File_Descriptor` with new methods.

```
class File_Descriptor_impl : virtual public File_Descriptor_skel {
private:
   char fname[512];
   fstream fio;
   // list of references to other servers
   LinkedList<File_Descriptor_ptr> ll;
public:

   // constructor and other methods not shown.
       ....

   void open(const char *fname)  {
      if(islocal(fname)) {  // check if file is located locally
       fio.open(fname, ios::in|ios::out);
      }
      else {
         for(ll.begin();ll.isendoflist(); ll.next()) {  // iterate through known servers
            if(ll.current()->islocal(fname)) {
               ll.open(fname);
            }
         }
   }

   CORBA::Boolean islocal(const char *fname) {
      // Checks if file is located on local file system (not shown)
   }

   void addserver(File_Descriptor_ptr p) {
       ll.add(p);
   }

};
```

Figure 13: CORBA (MICO) DFS server extensions using implicit binding (constructor and other methods remain same as in Figure 9).

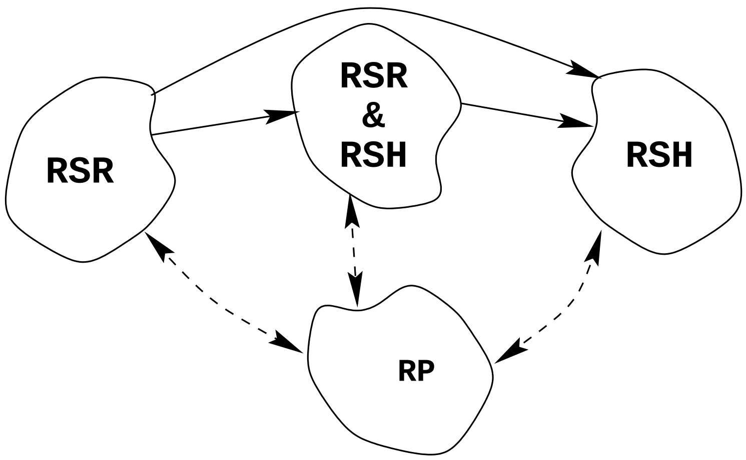

Figure 14: Sample interactions among RSR (Remote Service Requester), RSH (Remote Service Host), and RP (Registry Program). Note that a program can act both as an RSR and an RSH. The solid lines indicate invocations; the dashed lines imply the low-level communication with the RTS for register and bind.