Designing an Agent Synthesis System for Cross-RPC Communication

Yen-Min Huang and Chinya V. Ravishankar, *Member, IEEE*

Abstract—**Remote procedure call (RPC) is the most popular paradigm used today to build distributed systems and applications. As a consequence, the term "RPC" has grown to include a range of vastly different protocols above the transport layer. A resulting problem is that programs often use different RPC protocols, cannot be interconnected directly, and building a solution for each case in a large heterogeneous environment is prohibitively expensive. In this paper, we describe the design of a system that can synthesize programs (RPC agents) to accommodate RPC heterogeneities. Because of its synthesis capability, our system also facilitates the design and implementation of new RPC protocols through rapid prototyping. We have built a prototype system to validate the design and to estimate the agent development costs and cross-RPC performance. Our evaluation shows that our synthesis approach provides a more general solution than existing approaches do, and with lower software development and maintenance costs, while maintaining reasonable cross-RPC performance.**

Index Terms—**Heterogeneous RPC, RPC agent synthesis, RPC run-time**

I. Introduction

REMOTE procedure call (RPC) [6] is perhaps the most popular paradigm used today to build distributed applications. Many RPC semantics have been designed and implemented in recent years to meet application-specific requirements. Examples are synchronous RPC [6], [7], [8]; asynchronous RPC [9], [10], [11]; fault-tolerant RPC [12]; broadcast RPC [7], [8], [13]; maybe RPC (no-return RPC) [7], [8], [13], [14]; RPC with atomic transactions [15]; and RPC with a call-back mechanism [8], [13]. With emerging applications like multimedia conferencing and distributed real-time applications, it is conceivable that even more RPC protocols will be designed and implemented. Because of this diversity of RPC protocols, we adopt a general view of RPC as a protocol above the transport layer in this paper.

The problem with having many different RPC protocols is that user programs built on top of different RPC protocols cannot be interconnected directly. This difficulty not only greatly reduces the availability of software and resources in a large heterogeneous distributed environment but also increases the costs of developing and maintaining distributed applications with multiprotocol support. For example, this difficulty may arise when developers wish to build a multicast heterogeneous RPC to support fault tolerance, to construct a server accepting requests from many different RPC protocols, or to construct a client querying different name servers. Thus, the goal of this work is to design a system that supports cross-RPC communication (heterogeneous RPC) in a large heterogeneous distributed environment in which many systems are built using different RPC protocols.

The easiest way, and sometimes the only way, to perform cross-RPC is to introduce intermediaries (RPC agents) to facilitate communication between clients and servers. This method requires no changes to existing software. Building these RPC agents is hard, however, because it requires extensive knowledge of RPC and network protocols, and is time-consuming. In a large heterogeneous environment, the problem is exacerbated because there are too many RPC protocols, and building a solution for each case is prohibitively expensive.

We believe that a good cross-RPC solution in a large heterogeneous environment should meet the following criteria:

- **Economy**: It should require as little software development and maintenance effort as possible, because we want to minimize the effort of introducing a new RPC protocol.
- **Diversity**: It must accommodate as many different RPC protocols as possible, because we want to handle both existing RPC protocols and future RPC protocols.

Meeting these two criteria allows us to introduce new RPC protocols easily, and software using these new RPC protocols can be made available quickly. Also, the evolution of existing RPC protocols can be supported well.

A. Agent Synthesis

An agent synthesis scheme is a solution that can meet our design criteria well. An agent synthesis scheme uses a synthesizer to generate implementations of RPC agents from high-level descriptions. It is attractive because much of the effort of coding agents can be saved. Also, there are few restrictions on what kinds of RPC agents can be described and generated. Therefore, if designed properly, a synthesis scheme can provide a more general solution than can existing approaches [1]–[5], and with much lower agent development and maintenance costs. There is a major difference between our agent synthesis scheme and others: In addition to traditional stubs, we also generate implementations of RPC protocol machines as a part of the RPC run-time. In

Manuscript received May 1992; revised November 1993. This work was supported in part by the Consortium for International Earth Sciences Information Networking. Recommended by J. Zahorjan.

Y.-M. Huang was with IBM Corp., Research Triangle Park, NC. He is now with the Department of Electrical Engineering and Computer Science, University of Michigan, Ann Arbor, MI 48109–2122.

C. V. Ravishankar is with the Department of Electrical Engineering and Computer Science, University of Michigan, Ann Arbor, MI 48109–2122.

IEEE Log Number 9215570.

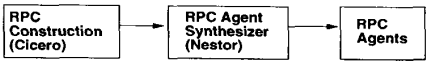

Fig. 1. The RPC agent synthesis scheme.

other words, for each different RPC protocol, a different RPC run-time may be generated along with the necessary stubs.

Broadly speaking, our RPC agent synthesis scheme (see Fig. 1) has two components: a language (Cicero) to describe RPC protocol constructions and a run-time environment (Nestor) to synthesize and activate RPC agents automatically. In this paper, we focus on the design of the synthesis scheme and its run-time environment.

Because of its synthesis capability, our system can also be used as a rapid prototyping tool for experimenting with new RPC protocols. This can be a very useful capability for developers. Therefore, our design considers this usage as well. In summary, our system is designed to provide developers with the following two services:

1) **Cross-RPC Service:** Supporting cross-RPC communication to increase software availability.
2) **Customized RPC Service:** Fast prototyping of customized RPC protocols for experimenting with new RPC features and semantics.

The rest of this paper is organized as follows. Section II describes the design of our RPC agent synthesis scheme. Section III describes the run-time support in Nestor. Section IV describes a real RPC protocol using Cicero. Section V describes the validation of the design by evaluating agent development costs and cross-RPC performance of synthesized agents. Section VI discusses related work. Finally, Section VII presents our conclusions.

II. Designing an RPC Agent Synthesis Scheme

Although the RPC agent synthesis scheme is straightforward (see Fig. 1), designing an RPC agent synthesis scheme is not easy, because many design issues must be considered and trade-offs carefully balanced. The design of the RPC agent synthesis scheme is the focus of this section. We first motivate the scheme by considering how the scheme and agents will be used (Sections II.A and II.B). Then we discuss how different RPC heterogeneities are handled (Section II.C).

A. Agent Synthesis Scenarios

Two agent synthesis scenarios are illustrated in Fig. 2(a) and 2(b). Fig. 2(a) illustrates the case where client and server programs may be modified. In this situation, RPC agents may be linked into user code. Fig. 2(b) illustrates the agent synthesis scenario where client and server programs may not be modified. The synthesized agents are independent processes in this case. Fig. 2(a) is the most likely scenario for customized RPC service, and Fig. 2(b) applies to cross-RPC service.

Both synthesis scenarios require a *link protocol* to connect the client and server agents. This link protocol is generated by our synthesis scheme. For customized RPC service, the link protocol represents the specified RPC protocol. For cross-RPC service, the link protocol usually implements cross-

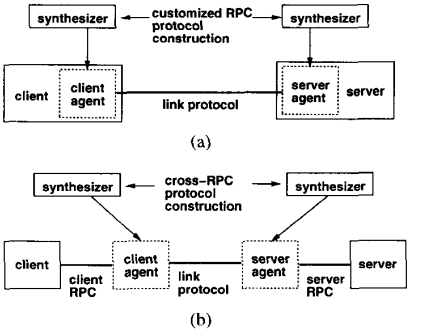

Fig. 2. (a) Customized RPC service. (b) Cross-RPC service.

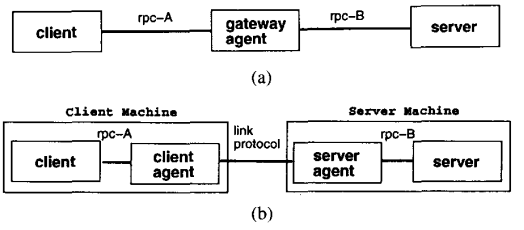

Fig. 3. (a) One-agent configuration. (b) Two-agent configuration.

RPC semantics representing the largest common denominator between two RPC protocols. These semantics are the best that can be achieved, because the two processes at the end of the cross-RPC link assume their local RPC semantics in their dealings with the world. We have no control over them or over the semantics of their native RPC protocols. The link protocol may also include some user-specified RPC semantics in addition to the shared RPC semantics. This flexibility allows users to tailor existing RPC protocols for special environments or applications. For example, in implementing a heterogeneous distributed transaction system, a user may wish to log each RPC argument and the results for crash recovery. This function can be implemented by the link protocol if neither the client nor the server RPC protocol provides this function.

B. Agent Configuration

The best agent configuration for customized RPC service is a two-agent configuration with the agents linked into the client and server (see Fig. 3). However, there are two possible configuration choices for cross-RPC service: a one-agent configuration and a two-agent configuration (see Fig. 3(a) and 3(b)). The one-agent configuration consists of a gateway agent that interconnects two programs using different RPC protocols (see Fig. 3(a)). The two-agent configuration can be constructed by splitting the gateway agent into two agents connected by a link protocol (see Fig. 3(b)). These two agents, the client and the server agent, are placed on the client and server machines, respectively.[1]

We use the two-agent configuration for our synthesis scheme because it results in a much cleaner synthesis scheme than does

[1] Placing agents on client and server machines is an access control issue, not a limitation of the scheme.

the one-agent configuration. It is cleaner because each agent needs to know only the local and the link RPC protocols, and the run-time support for both the native and the link RPC protocols are locally available. In contrast, a one-agent configuration does not have all of the run-time support available locally, because the single gateway agent must be located on either the client machine or server machine. In either case, it must be aware of the details of the RPC that are not available locally, complicating agent construction and synthesis. Further, it is easy to handle both customized RPC service and cross-RPC service with the two-agent configuration, because customized RPC service must use a two-agent configuration.

To examine the performance implications of a two-agent configuration, we use the following equations to determine S, the slowdown in performance with respect to a reference RPC *ref*:

$$S = \frac{T_{\text{total}}}{T_{\text{ref}}} = \frac{T_{\text{client}} + T_{\text{link}} + T_{\text{server}}}{T_{\text{ref}}}. \quad (1)$$

T_{total} represents the total elapsed time for our RPC, which is the sum of the elapsed time for passing data from a client to the client agent (T_{client}), from the client agent to the server agent (T_{link}), and from server agent to the server (T_{server}). The slowdown S is defined as the ratio of the total elapsed time of our RPC to T_{ref}, the elapsed time of the reference RPC (the RPC we are compared with).

Surprisingly, using a two-agent configuration causes little degradation of RPC performance for most cases. For the customized RPC case, performance is unaffected, because both agents can usually be linked with the corresponding client and server programs directly. This fact is expressed by the following conditions:

$$T_{\text{link}} \approx T_{\text{ref}}$$
$$T_{\text{link}} \gg T_{\text{client}}, T_{\text{server}}.$$

The first condition assumes that the link protocol is well implemented and has performance competitive with the reference RPC. This assumption will be supported by the performance data in Section V-B. The second condition represents the fact that local procedure calls are much faster than remote procedure calls. With these two conditions, (1) reduces to the following:

$$S = \frac{T_{\text{client}} + T_{\text{link}} + T_{\text{server}}}{T_{\text{link}}} = 1 + \frac{T_{\text{client}} + T_{\text{server}}}{T_{\text{link}}} \approx 1.$$

In some cases, using knowledge about the client and server machines, it may be possible to build customized RPC whose performance may even exceed the performance of the native RPC system. For example, if it is known that the client and server machines both use the same data representation and compiler, the programmer can synthesize agents that bypass the marshalling/unmarshalling routines. For an RPC that supports only one external data representation (like SUN RPC/XDR), however, such a bypass cannot be accomplished without changing the RPC protocol.

For cross-RPC, the slowdown is determined by the ratio of the elapsed time of the native RPC ($T_{\text{client}}, T_{\text{server}}$) to the elapsed time of the link protocol (T_{link}). If the client and the server hosts are connected through a wide-area network, we will have slowdown ratios close to 1.0, because the two conditions mentioned above are still valid (i.e., network delay is the dominating term in T_{total}). In the worst case, when the client and the server hosts are on the same local area network, and $T_{\text{link}} \approx T_{\text{client}} \approx T_{\text{server}}$, the slowdown ratio may be as high as 3.0. The absolute value of the elapsed time in these cases may still be acceptable, however. For example, we believe that a slowdown in communication from 2 ms to 6 ms is quite tolerable and should not cause problems in most cases. In addition, for most cross-RPC cases, programmers have control over the client program code and can link the synthesized agent with the client program directly. In these cases, the scheme degenerates to the one-agent configuration, and the performance becomes even more acceptable. Therefore, we conclude that the slowdown in two-agent configuration is acceptable and is a reasonable price to pay for a clean design and implementation.

C. Handling RPC Heterogeneities

One crucial issue in designing an RPC agent synthesis scheme is determining how different RPC heterogeneities should be handled. We have found that a proper classification of RPC heterogeneities to be very useful in providing insights on how to handle various RPC heterogeneities.

We define an RPC protocol simply as a protocol above the transport layer. Although the general scheme discussed here can be applied to other layers, we limit our discussion to the heterogeneity issues above the transport layer. For accommodating heterogeneities at the transport layer, readers can refer to [2] for more information.

Two RPC systems can be very different. Differences may exist in the call semantics,[2] in the failure semantics,[3] in the RPC topology,[4] in the external data representation,[5] in the naming and binding mechanism,[6] in the authentication/encryption mechanism,[7] and so on. Clearly, with so many heterogeneities to be accommodated, building a solution for each case is prohibitively expensive. Even using a synthesis scheme, we must minimize the amount of code to be synthesized to make the scheme manageable.

Our approach is to first classify RPC heterogeneities into those that are *semantics-dependent*, and those that are *semantics-independent*. For example, heterogeneities in call semantics and failure semantics are semantics-dependent, whereas heterogeneities in RPC message format are semantics-independent because they are artifacts of implementation.

We handle RPC heterogeneities differently depending upon their type. Semantics-dependent heterogeneities are

[2] Examples are synchronous call semantics (SUN RPC) and asynchronous call semantics (ASTRA).

[3] Examples are at-most-once (HP/Apollo NCA RPC) and exactly-once (ARGUS).

[4] Examples are one-client-one-server (Xerox Cedar RPC) and one-client-many-server (SUN broadcast RPC).

[5] Examples are NDR (HP/Apollo NCA RPC) and XDR (SUN RPC).

[6] Examples are UUID/Location Broker (HP/Apollo NCA RPC) and (prog#, ver#)/Portmapper (SUN RPC).

[7] Examples are Grapevine Database/DES (Xerox Cedar RPC) and UNIX/DES (SUN RPC).

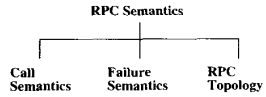

Fig. 4. Aspects of RPC semantics implementation.

TABLE I
LINES OF SOURCE CODE FOR DIFFERENT RPC IMPLEMENTATIONS

Function	DCE RPC	SUN RPC	Minimal SUN RPC Client Agent
BASIC	5549 (21.2%)	342 (6.6%)	342 (39.8%)
CONV	4803 (18.4%)	933 (18.2%)	179 (20.8%)
CLNT	3852 (14.7%)	1212 (23.7%)	185 (21.5%)
NS	4899 (18.8%)	494 (9.6%)	119 (13.8%)
AUTH	2202 (8.4%)	1082 (21.1%)	35 (4.1%)
SV	3906 (15.0%)	1008 (19.7%)	0 (0.0%)
OTHER	918 (3.5%)	50 (1.0%)	0 (0.0%)
Total	26129 (100.0%)	5121 (100.0%)	860 (100.0%)

handled by synthesizing the implementation of the specified semantics directly, and semantics-independent heterogeneities are handled by providing a default implementation of each mechanism. For performance reasons, programmers must be able to control the implementation of RPC semantics. Therefore, we use a synthesis approach designed to give programmers the maximum flexibility in describing their RPC semantics implementation. Fig. 4 summarizes three aspects of RPC semantics: call semantics, failure semantics, and RPC topology, which must be described in the protocol construction language and synthesized by the synthesizer.

Semantics-independent heterogeneities are encapsulated in the link protocol and have little effect on applications. For example, users need not be concerned with what external data representation is used, as long as the link protocol provides one. Therefore, all mechanisms pertaining to semantics-independent heterogeneities are provided through libraries or as run-time services. Because there is no need to describe semantics-independent implementations, our construction language need describe only RPC semantics-dependent implementations. This not only makes the construction language simpler but also greatly reduces the complexity of the synthesis scheme, with little or no detriment to the generality of the solution.

To quantify the reduction in complexity due to this RPC heterogeneity classification, we classify the source code[8] of different RPC implementations (OSF/DCE and SUN) into several categories, based on the common functionalities provided by the RPC run-time. This classification provides information about the complexities of implementing or synthesizing each functionality. Results are listed in the first two columns of Table I.

[8]The source code consists of both C source files and related include files. The number of lines of source code is estimated by counting ';'. The number of lines of include files is estimated by counting ':' and '#'.

The categories listed in Table I are defined as follows.

- BASIC: Routines for implementing the common data structures, the basic utilities, and the infrastructure of the run-time.
- CONV: Routines related to data representation conversion.
- CLNT: Routines to implement the client protocol machine.
- NS: Routines related to name service.
- AUTH: Routines perform encryption/decryption and authentication.
- SV: Rroutines to implement the server protocol machine.
- OTHER: Routines not belonging to any of the above categories (e.g., debugging related routines).

The last column in Table I deserves some additional explanation. This column estimates the effort of hand crafting a client agent that provides minimal SUN RPC functionality. This minimal client agent implements SUN RPC at-most-once semantics on top of TCP.[9] The implementation of the client agent also includes the basic functionalities of, for example, marshalling and server binding. The estimate in Table I was arrived at by extracting related source code from the current SUN RPC implementation.

From Table I, we can see that it is too complex to synthesize an entire RPC run-time. This is because there are simply too many different aspects and details to be described, such as RPC protocol machines, the naming scheme, the security mechanisms, and data representation.

If we synthesize only the code related to semantics-dependent heterogeneities (CLNT + SV), however, more than 60% of code can be provided by either the run-time or libraries. In other words, the maximal amount of code to be synthesized (CLNT + SV) is 30% to 40% of the entire RPC package, and includes all of the different RPC semantics supported by a specific RPC package. Luckily, most of the time, only one of the many supported RPC semantics is used, and usually only a client agent need be synthesized.[10] In such cases, the amount of code to be synthesized can be further reduced to 4% ($= \frac{185}{5121}$) of an entire RPC run-time. Nevertheless, the CLNT code still accounts for more than 20% of the entire client agent code. Therefore, we would like to develop a language to further reduce the coding effort for implementing RPC protocols. We have developed Cicero, a protocol construction language, is developed for describing RPC protocols. We will only briefly describe Cicero in this paper, however, because the semantics of Cicero language constructs are issues orthogonal to the synthesis scheme and are described elsewhere [16].

III. NESTOR: RUN-TIME SUPPORT FOR AGENT SYNTHESIS

The major difference among Nestor, our run-time, and traditional RPC run-times is that Nestor provides additional support for facilitating agent synthesis. Therefore, we focus

[9]It would require more code if the client agent were built on top of UDP.

[10]The server and the server agents have already been built.

on details about how our agent synthesis scheme works and how Nestor supports agent synthesis.

A. Specifications and Agent Synthesis

Synthesizing agents involves two steps:

1) constructing necessary synthesis specifications, and
2) synthesizing agents from specifications.

The first step involves describing RPC semantics, interfaces, and instructions for synthesis. The second step involves generating code, compiling, and linking all of the components and libraries to create executable images of agents.

To synthesize an RPC agent, three specifications are required: the RPC protocol construction, the RPC interface specification, and the RPC agent profile specification. The RPC protocol construction and the RPC interface together determine what agent will be synthesized. Specifically, the RPC protocol construction (written in Cicero) describes the implementation of RPC semantics (i.e., call semantics, failure semantics, and RPC topology). The RPC interface specification describes the remote interface specification and is used to generate stubs, which are used to interface with the client, the server, and our run-time libraries. The agent profile specification determines how an agent will be synthesized and managed. The agent profile specification is a configuration file, containing instructions for synthesizing and managing agents. For example, the agent profile specification defines the synthesis environment and activation parameters for an agent. For each protocol, two sets of these specifications are needed: one for the client agent and one for the server agent.

Nestor uses a set of libraries and utility programs to synthesize executable images of RPC agents. These libraries include the protocol construction library and the external data representation library. The protocol construction library provides the functions to implement the link protocol between two agents. The external data representation library provides the marshalling/unmarshalling routines for RPC agents. The utility programs used by Nestor consist of compilers for Cicero and C, a stub generator, and a software packaging utility (like the UNIX *make*).

Fig. 5 illustrates how these specifications and utility programs work together to synthesize an agent. The Cicero compiler compiles the RPC protocol construction and outputs a C-code implementation of the specified RPC semantics. This C code will be compiled by the native C compiler and linked with the libraries to implement the link protocol. The stub generator compiles the RPC interface specifications and generates the stub routines that interface with the client or server program and with the link protocol implementation. For customized RPC service, Nestor provides its own stub generator and library to the user. For cross-RPC service, Nestor expects stub generators from the native RPC facilities, which it uses to synthesize the native RPC stub for the agent. When a stub generator is not available, users are required to provide RPC stubs. Finally, RPC stubs, libraries, and the link-protocol implementation are linked together to form an agent. This entire synthesis process is specified in the RPC agent profile specification and controlled by the software packaging utility.

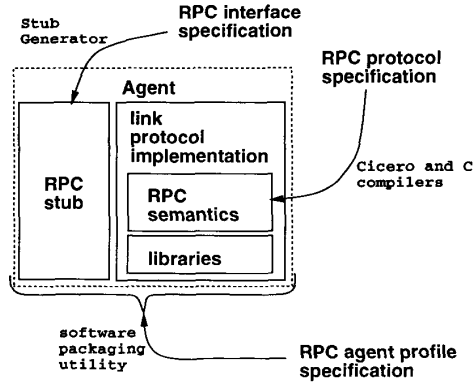

Fig. 5. Using the synthesis support to synthesize an agent.

B. Other Related Support for Agent Synthesis

There are two other kinds of support related to agent synthesis: the specification-transfer support and the agent-management support. The specification-transfer support facilitates importing or exporting protocol constructions between sites, and is useful because the protocol constructions may not be available at the machine where an agent will be synthesized. For example, a user may wish to perform a heterogeneous RPC using server RPC semantics, and the client-agent construction for the server RPC protocol may not be available at the client machine. To synthesize the client agent, the client can import the client-agent construction from the server. This support is provided to encourage sharing of RPC protocol constructions in a large heterogeneous environment, so that programmers can use or customize existing protocol constructions instead of writing new ones themselves.

The ability to import protocol constructions from the outside not only reduces agent development costs but also offers other advantages. It provides immediate software availability after a protocol construction is created or updated. Clients would import the new specifications and synthesize local agents. It minimizes disturbance when updating existing RPC's and introducing new RPC's. Hence, RPC protocol evolution is well supported. It also offers the opportunity to synthesize specialized code to improve performance. Finally, it also makes the synthesis solution scalable, and makes each site fully autonomous.

The agent management support is responsible for all activities related to agent management, including agent run-time activities and agent caching. At run-time, the agent-management support is responsible for controlling the activation, execution, and termination of agents. All of these activities are specified in RPC agent profile specifications. For example, users can provide the activation instructions for a newly synthesized agent in the RPC agent profile specification, so that Nestor can automatically activate the synthesized agent. If an agent is linked with a client or server program, the agent management support is used to activate the client and the server directly.

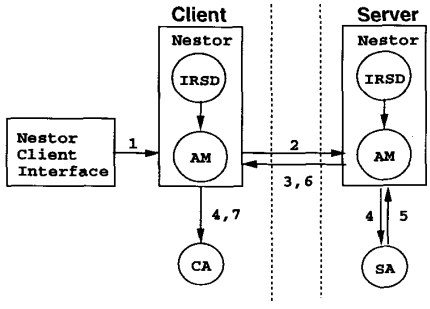

Fig. 6. The agent synthesis process in Nestor.

The agent management support can also be instructed to cache synthesized agents for future use to avoid synthesizing frequently used agents repeatedly.

C. An Agent Synthesis Scenario

To describe how Nestor synthesizes agents, we will present a cross-RPC service scenario where agents are synthesized and activated automatically. We also assume that the user has already discovered the server host address through the Nestor name service support (see Section III.D for details). The steps in the agent synthesis process are shown as numbered arcs in Fig. 6.

The Nestor run-time environment consists of two components: an Internet RPC Service Daemon (IRSD), and an agent manager (AM). IRSD is a process that handles all synthesis requests and is brought up at machine initialization time. It initializes itself by reading files containing configuration information and the specifications of services exported from the site. It then waits for requests from local clients and remote IRSD's. Upon receiving a request, IRSD forks off a copy of the Agent Manager (AM) to serve the request. The AM is responsible for synthesizing, executing, and terminating an agent. To facilitate interaction between the user and Nestor, the user is provided with a command-line interpreter called the Nestor Client Interface. It allows the user to interact with Nestor by issuing commands. Here we assume that the user uses this interface to contact Nestor.

Initially, Nestor runs as an IRSD daemon on the local machine and listens on well-known ports. When the user first contacts the local Nestor instance, it creates an AM to handle the user's requests. The user issues the synthesis request to the client AM (step 1). The client AM locates the client-agent synthesis specifications and contacts the server-side Nestor instance (step 2). The server-side Nestor instance now forks off an AM to handle the requests from the client AM. After the server AM verifies the client's requests, both the client and the server AM synthesize the agents (steps 3 and 4). After the server agent is synthesized, the server agent is activated. The port number used by the server agent is returned to the client agent through AM's (steps 5 through 7). Now the agents are ready to perform the specified heterogeneous RPC.

D. Other RPC-Related Support

Name service is also provided by Nestor. In heterogeneous distributed environments, name service is an important research topic by itself. It is not the focus of the current design, however; therefore, for our system, we simply apply existing mechanisms as appropriate for our purposes.

The Nestor name service support helps a client contact a server by using two items of information: the server host address and the port number of its agent. The server host address is discovered by querying a global database[11] that has knowledge of all available services in the network. The port number is obtained through cooperation between the client-side and the server-side Nestor instances. More specifically, the server-side Nestor instance obtains the port number exported by the server agent and passes it to the client-side Nestor instance. The client agent can now obtain the port number from its local Nestor instance.

The Nestor name-service support is a default name-service mechanism provided to bypass heterogeneity problems in name service. For cross-RPC communication, different naming mechanisms may be used for the client and the server RPC systems. In our scheme, the differences in naming mechanisms are subsumed by RPC agents, because the client and the server always contact their agents by using the native RPC run-time support. How the client agent locates the server agent is independent of the native naming mechanisms. Thus, Nestor provides its own name service support to locate agents without interfering with the native naming mechanism. This is advantageous because no explicit mapping is necessary between the native naming model and the Nestor naming model. The Nestor name service support exemplifies how we handle semantics independent heterogeneities (see Section II.C).

IV. An Example

The example given in this section serves two purposes. First, it illustrates how to use Cicero to construct an RPC protocol. Second, it provides a basis for estimating agent development costs using our scheme, which is discussed in Section V.

The example uses Cicero to construct a synchronous multicast RPC protocol with at-least-once failure semantics. That is, a client can contact more than one server at a time by making one remote procedure call, and the caller is blocked until all of the replies are received or until an error occurs. We implement these semantics in two steps. First, we describe the at-least-once semantics. Then we add the code for the call semantics and the topology.

Let us assume that the original protocol specification for the at-least-once semantics are specified by an extended finite state machine (FSM) shown in Fig. 7, where input events can be associated with conditions.

To prepare readers for the example, we first briefly introduce Cicero and its constructs. Cicero is an event-driven specification language derived from POST [17]. Unlike specifications in languages like LOTOS [18] and Estelle [19], which are declarative (nonexecutable), Cicero specifications are constructive (executable). This feature allows programmers to provide execution information to guide the protocol synthesis

[11] It does not matter whether the database is distributed or replicated. Here we treat it as a single entity.

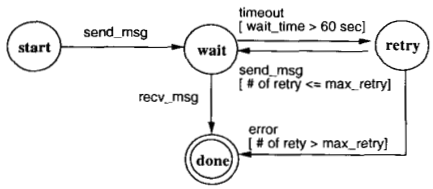

Fig. 7. An extended FSM diagram for at-least-once semantics.

process in generating efficient implementations. Cicero has five constructs: **emit, when, cond, bundle,** and **escape**. The **emit** construct is used to generate event instances. Each **when** construct represents one thread of control and can trigger actions each time that specified events are observed. The **cond** construct implements conditional branches. The **bundle** is a modularization construct similar to the procedure and is invoked synchronously. The **escape** construct is used to include C statements in Cicero by enclosing them in "{" and "}."

Although the Cicero specification for the at-least-once semantics can be more compact, we present a version that is slightly longer, in order to make the implementation easier to understand. There is a one-to-one mapping in events between the FSM specification in Fig. 7 and the Cicero code segment, except that the *send_msg* event is replaced by a library call. The correspondence between the Cicero code segment and the original specification is indicated within the comments of the code segment. Table II describes the semantics of various event syntax associated with the **when** constructs in the Cicero code segments.

```
1 bundle client_rpc(CC_handle_t handle,
  CC_msg_t *msg)
2 {
3     int       err_code;
4     long      wait_time;
5     event     recv_msg, wait, retry;
6
7     when (INIT): /* FSM: start -> wait */
8     {
9     wait_time = 60; /* wait
            for 60 sec. */
10    CC_set_udef_sendmsg(handle,
            msg);
11    CC_ioctl(handle, RECVBLOCK,
            TRUE);
12    CC_send_udef_msg(handle);
            /* send_msg */
13    }
14    emit recv_msg;
15    emit wait;
16    end;
17    when (recv_msg): /* FSM: wait
      -> done */
18          { err_code = CC_recv_udef_msg
            (handle);}
19          emit return:(val=err_code);
20    end;
```

TABLE II
SEMANTICS OF VARIOUS **WHEN** CONSTRUCTS
$|x|$ = NUMBER OF OCCURRENCES OF EVENT x TO DATE

Syntax	Description
when (x): A end;	executes action A when x occurs.
when (x?i):A end;	same as the above, with variable $i = \|x\|$
when (x)*N: A_1 end: A_2	if $\|x\| < N$ executes A_1 else executes A_2
when (INIT): A end;	A is the first executed action when enclosing **bundle** is invoked.

TABLE III
DESCRIPTION OF FUNCTIONS USED IN BUNDLE CLIENT_RPC()

Function	Description
CC_send_udef_msg	sends out an RPC message.
CC_recv_udef_msg	waits for an RPC reply message.
CC_wait	pause for a period of time before continuing.
CC_ioctl	set input/output control options (similar to UNIX *ioctl*()).
CC_set_udef_sendmsg	associates an RPC message with the communication handle, so that it can be sent out later.

```
21    when (wait): /* FSM: wait -> retry */
22          { CC_wait(wait_time); }
23          emit retry;
24    end;
25    when (retry)*MAX_RETRY: /* FSM:
      -> retry wait/err */
26          { CC_send_udef_msg(handle); }
            /* send_msg */
27          emit wait;
28    end: emit return:(val=E_RPCFAIL);
      /* rpc failed */
29}
```

After sending out the message (line 12), two **when** constructs (lines 17 and 21) run concurrently, waiting for a reply or a time-out, respectively. If a reply is received, the **bundle** returns. If a time-out occurs, the original message is sent again. Such retry continues until either a reply is received or the number of retries exceeds the limit MAX_RETRY. In the later case, the **bundle** returns with an error. All of the functions with prefix name "CC_" are provided by the Cicero communication library, and their functionality is briefly described in Table III. The Cicero communication library is derived from our universal RPC toolkit [20], which is a toolkit for prototyping a variety of RPC systems rapidly.

We now complete the description of the synchronous multicast RPC and specify the synchronous call semantics and multicast topology. The description is listed below.

One multicast RPC is broken into a number of component RPC's, one for each server binding (represented by handles). Each component RPC (client_rpc()) runs with its own thread and has the at-least-once semantics described previously (lines 12–18). Each time a component RPC completes, a *reply* event instance is emitted (line 16). If enough *reply* instances have been collected, the multicast RPC completes (lines 19–24). If

TABLE IV
CODING EFFORT SAVING FOR CLIENT AGENTS

Case (language)	BASIC	CONV	CLNT	NS	AUTH	SV	OTHER	Total
Handcrafted client agent (C)	342	179	185	119	35	0	0	860
	39.8%	20.8%	21.5%	13.8%	4.1%	0.0%	0.0%	100.0%
Synthesized client agent (Cicero)	0	0	35	0	0	0	0	35

any error occurs, the multicast RPC returns with an error (line 21).

```
1   bundle sync_multicast_rpc(
2           int num; /* num. of
            handles */
3           CC_handle_t handle_array[];
                /* handles */
4           CC_msg_t *msg /* message
            to be sent */ )
5   {
6       event   rpc, reply;
7       int     ret;
8
9       when (INIT):
10          emit rpc; /* the 1st rpc */
11      end;
12      when (rpc?i):
13          cond (i < num):
14              emit rpc; /* invoke the
                another simple rpc */
15              ret = client_rpc
                (handle_array[i-1],msg);
16              emit reply:(val=ret);
17          end; /* i > num: do nothing */
18      end;
19      when (reply?j):
20          cond (reply.val != OK):
            /* error */
21              emit return:(val=
                reply.val);
22              (j = num): /* multicast
                rpc completes */
23              emit return:(val=OK);
24          end;
25      end;
26  }
```

V. EVALUATION

To validate our design, we evaluate our synthesis solution from two aspects: agent development costs and cross-RPC performance using synthesized agents. We have built a prototype for both Nestor and Cicero and have ported them to UNIX BSD 4.3 and Mach 2.5.

A. Agent Development Costs

We use lines of source code to estimate the costs of developing agents. In the above example, it takes 35 lines[12] to describe a synchronous multicast RPC. From Table I in Section II.C, it takes 860 lines of C code to handcraft a simple SUN RPC client agent, which has no multicast capability. These two cases are summarized in Table IV.

Because the example given above implements more complex RPC semantics than does the client agent described in Section II.C, it takes even less Cicero code to describe the same RPC semantics implemented by the client agent. Therefore, we can estimate confidently that more than 95% of coding effort can be saved by using our agent synthesis scheme.

Although the 95% saving may sound dramatic, it is not surprising, because our heterogeneity classification allows us to provide 80% of agent code through libraries and run-time services for semantics-independent mechanisms. The remaining 15% is generated by the Cicero compiler or provided by the Cicero run-time library. Similar savings can also be found for the corresponding server agent, because the implementations of the client and server agent are symmetrical and are comparable in their code sizes. (See CLNT and SV in Table I.)

There are two other factors in our scheme that can further reduce agent development costs. First, a large portion of Cicero code can often be written to mirror some existing protocol specification, making it easier to construct correct protocol implementations. We have illustrated this process in the above example, translating an extended FSM protocol specification into Cicero code segment. Second, the effort of coding agents can be eliminated altogether if the agent constructions are available elsewhere on the network. Programmers can use Nestor's specification-transfer support to obtain these constructions.

B. Cross-RPC Performance

Cross-RPC performance depends on the performance of native RPC's and the link protocol implemented by synthesized agents. Since we have no control over the performance of native RPC's, we focus on the performance of synthesized agents. Agent performance depends on two factors: the Cicero run-time overhead and the performance of the communication primitives provided by the Cicero communication library. The Cicero run-time overhead is about 0.3 ms on average, which mostly comes from the overheads of thread management and mutual exclusion support [16].

The performance of the communication primitives is represented by the performance of an RPC protocol constructed using these primitives. These RPC protocols, ATM1–RPC/UDP and ATM1–RPC/TCP, are constructed on top of UDP and TCP, respectively, with at-most-once semantics. The perfor-

[12] To facilitate comparisons with the results in Table I, Section II.C, the number of lines of source code was also computed by counting ";".

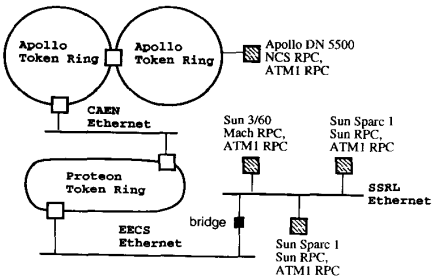

Fig. 8. The network environment for the performance measurement.

TABLE V
NULL RPC PERFORMANCE

Transport	SUN RPC	ATM1-RPC	S_{sun}
UDP	2.58±0.01ms	2.29±0.04ms	0.88
TCP	3.33±0.02ms	2.58±0.04ms	0.77

mance data measured were the round-trip elapsed time for null RPC under lightly loaded network and workstations, including language run-time overheads. The null-RPC performance of SUN RPC/UDP and SUN RPC/TCP are provided for comparison. (See Table V.)

As Table V indicates, our RPC performance is competitive, even with the Cicero overheads included. The range of RPC performance is about 2.5 ms and up. Adding this performance with the average language overhead (≈0.3 ms), the performance is still competitive with SUN RPC. With normal daily workloads on the network and workstations, the round-trip elapsed time will be even longer (5 ms ~ 10 ms), making the language overhead even smaller in relative terms.

Two sets of measurements were carried out for cross-RPC: one set each for a one-agent configuration and for a two-agent configuration. Each set of measurements consisted of three cross-RPC cases involving SUN RPC, Mach RPC, and HP/Apollo NCS/NCA RPC, respectively. These RPC's all passed a one-character string to their servers and get back an integer. Both sets of measurements used the ATM1-RPC/TCP as the link protocol.

The machines and the RPC's used in these measurements are illustrated in Fig. 8 as shaded boxes. Specifically, the Sun Sparc 1 workstation used SUN RPC; the Sun 3/60 used Mach RPC; and the HP/Apollo DN5500 used NCS RPC. These machines were connected through our campuswide network (Fig. 8). In measuring the one-agent configuration cases (Table VI), a Sun Sparc 1 workstation was used as the client machine running ATM1–RPC/TCP in all three cases. This machine was also used as the client machine running SUN RPC in the two-agent configuration cases (Table VII).

Table VI lists the cross-RPC performance for three one-agent configuration cases with an RPC agent running at the server machine. The first two are cross-RPC cases (ATM1/SUN and ATM1/Mach) on our local Ethernet, and the last case (ATM1/NCS) is cross-RPC through several gateways. For each case, a breakdown of remote ATM1 RPC and the local RPC overheads is provided for analysis. Depending on

TABLE VI
CROSS-RPC PERFORMANCE WITH ONE-AGENT CONFIGURATION

RPC1–RPC2	Remote ATM1	Local RPC	Cross-RPC	S_{atm1}
ATM1 – Mach	4.48±0.01 ms	1.38±0.04 ms	5.78±0.02 ms	1.29
ATM1 – SUN	2.58±0.04 ms	3.70±0.03ms	6.34±0.03ms	2.45
ATM1 – NCS	12.0±0.5 ms	5.1±0.1ms	19.9±0.5ms	1.65

TABLE VII
CROSS-RPC PERFORMANCE WITH TWO-AGENT CONFIGURATION

RPC1 – (ATM1) – RPC2	TCP Socket	Cross-RPC	S_{tcp}
SUN – (ATM1) – Mach	3.6±0.1ms	9.7±0.1ms	2.69
SUN – (ATM1) – NCS	11.1±0.5ms	21.4±1.0ms	1.94
Mach – (ATM1) – NCS	12.6±0.8ms	20.5±1.0ms	1.62

the relative performance between the remote ATM1 RPC and the local RPC, the cross-RPC slowdown over ATM1 RPC (S_{atm1}) ranges from 1.29 to 2.45.

Table VII lists the cross-RPC performance for three two-agent configuration cases. The first case is two-agent cross-RPC on our local Ethernet, and the rest are cross-RPC's across several gateways. For comparison, we also provide the round-trip time measurements for TCP sending a one-byte message directly between the client and the server. These measurements are used to approximate homogeneous RPC performance, and, because no RPC layer overhead is included here, the measurements give us a conservative estimate of a homogeneous RPC performance on top of TCP.

The two-agent configuration is the worst-case scenario for cross-RPC, when client and server programs may not be modified. In this worst-case scenario, the cross-RPC slowdown ranges from 1.6 to 2.7 over our estimated homogeneous RPC performance. Because the client and the server may not be modified in this case, however, handcrafted cross-RPC implementations do just as poorly. To compare with handcrafted implementations of one-agent configurations, one-agent cross-RPC's (Table VI) are used to approximate their performance. The two-agent cross-RPC is only slightly slower (3% to 7% slower) than the one-agent RPC in the cross-gateway cases. In the worst case, it is about 50% slower than the one-agent implementation. This happens when a remote RPC takes about the same amount of time as a local RPC, a somewhat unlikely scenario.

VI. RELATED WORK

Most related work can be categorized into four classes: protocol libraries/architecture, remote evaluation schemes, remote interface synthesis, and specification compilers. Our solution is a hybrid of these four classes of solutions.

In general, solutions based on protocol libraries or architecture provide programmers with a library interface and an architecture for constructing protocols. The interface and the architecture are engineered by factoring out common protocol functionalities and abstractions. HCS/HRPC [1], *x*-

kernel [21], [22],[13] and TACT [2][14] are examples of this type of solution. However, library architectures are restricted by the number of implementations provided in their libraries. Therefore, introducing a new RPC protocol often requires updates to every library on the network, a requirement that is not desirable in large heterogeneous distributed environments. Our solution resolves this difficulty by importing the remote protocol construction to synthesize a local agent that subsumes the heterogeneities and facilitates communication with the remote protocol. Our solution can easily incorporate existing libraries/architecture solutions as a special synthesis case that only requires packaging RPC agents using the available libraries.

Remote evaluation schemes require a client to pass a piece of code to the remote machine, where it is evaluated to provide service to the client. Although all remote-evaluation systems have similar code distribution mechanisms, the context and the language used in these systems are very different. For example, HDS/NCL [5] uses the special-purpose programming language NCL (based on LISP) to implement a heterogeneous network file system. Another example is REV [4], which is embedded in the programming language CLU for supporting general purpose remote evaluation. Because our goal is to accommodate RPC protocol heterogeneities, which is very different from the focus of such earlier work, our system provides programmers a special protocol construction language (Cicero) for describing RPC protocols. The protocol descriptions are then used to synthesize RPC agents.

Remote interface synthesis schemes generate communication code according to some interface definition, so that procedures/modules written in different languages can communicate with each other in a heterogeneous environment. Examples of this type of scheme are the POLYLITH software bus [23] and the Horus stub generator [24]. All of these solutions are focused on resolving heterogeneities in data types and data representations. They all provide mappings from local data types and data representation to predefined data types and representations. Although our system also uses this approach to resolve heterogeneities in data types and representations, it also allows programmers to synthesize RPC agents to interconnect programs by using different RPC protocols. This feature is absent in remote interface synthesis solutions.

The work on specification compilers [25], [26], [27], [28] has been focused on generating correct implementations from the formal protocol specifications, as in the LOTOS and Estelle languages. The portions of a protocol implementation that can be generated, however, depend on the environment and the specification language used, and the efficiency of generated code is also a concern [29]. Code quality is an important consideration, especially in the RPC context. Based on a protocol construction language, our prototype implementation can produce synthesized RPC agents with performance competitive with handcrafted RPC implementations. In some customized RPC cases, we believe that the synthesized code may even perform better than handcrafted solution. This apparent paradox arises because synthesis can always tailor code for each case to improve performance, whereas handcrafted solutions usually incur overhead, targeting the general case. A good example of this principle is seen in the Synthesis kernel [30], which provides significant speedup for UNIX kernel calls by generating specialized kernel routines for specific situations. In our context, for example, agents may bypass data conversion routines if both the client and the server use the same data representation.

[13] The x-kernel is included here because of its protocol synthesis capabilities.

[14] TACT is used for accommodating heterogeneities for Transport layer protocols, not for RPC. it is included here, however, for its library approach.

VII. Conclusion

Our work illustrates that an agent synthesis scheme is an effective method for dealing with the many instances of RPC heterogeneity in heterogeneous distributed environments. Our agent synthesis scheme provides a solution with low software development and maintenance costs while maintaining reasonable cross-RPC performance.

The two-agent configuration is key to the handling of large numbers of heterogeneity instances. This configuration allows us to concentrate on synthesizing agents to handle only RPC semantics-dependent heterogeneities, and to encapsulate the semantics-independent heterogeneities within a local agent through run-time support and libraries, resulting in an 80% saving of agent development costs (see Section V-A). This encapsulation also greatly reduces the complexity of the synthesis scheme. For example, it allows us to bypass the heterogeneity problems in name service, which would otherwise add another dimension of complexity to the synthesis scheme. The two-agent configuration also allows clients to import protocol constructions from outside, which can further reduce agent development costs and support evolution of RPC protocols.

The ability of our system to import protocol constructions from the outside provides immediate software availability after a protocol construction is created or updated. Clients would import the new specifications and synthesize local agents. It minimizes disturbance when updating existing RPC's and introducing new RPC's. Hence, RPC protocol evolution is well supported. It also offers the opportunity to synthesize specialized code to improve performance. Finally, it makes the synthesis solution scalable and makes each site fully autonomous.

It can be difficult to get performance on cross-RPC's equivalent to that of homogeneous RPC. This is because a slow native RPC can easily become a performance bottleneck, especially when the programs involved may not be modified. However, it is still much better to get work done with cross-RPC than to get no work done at all. More importantly, by using our synthesis scheme, cross-RPC can be accomplished with little software development and maintenance costs, and can achieve performance acceptable by most applications, even in the worst-case scenario.

We end with a comment on general cross-RPC's. Automatic synthesis works best for cross-RPC within the same class of RPC semantics, for example, cross-RPC among at-most-once RPC's or among at-least-one RPC's. Although cross-

RPC between dissimilar RPC classes is possible in general and is allowed in our scheme, it is clearly impossible to accomplish more than preserving the semantics common to the two RPC classes. The semantics of the local RPC's are often deeply embedded in the user code. Any solution that attempts to resolve all semantics mismatches probably requires modifications to client, server, or even native RPC run-time. Because it is impossible for us to determine the user-level semantics intended, it is the user's responsibility to make sure that the client and the server programs can be meaningfully interconnected and to provide the appropriate agent synthesis specifications to interconnect them.

REFERENCES

[1] 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 Eng.*, vol. 13, no. 8, pp. 880–894, Aug. 1987.

[2] J. Auerbach, "TACT: A protocol conversion toolkit," *IEEE J. Select. Areas Commun.*, vol. 8, pp. 143–159, Jan. 1990.

[3] Sun Microsystems, *Open Network Computing—RPC Programming*, Mar. 1991.

[4] J. W. Stamos and D. E. Gifford, "Implementing remote evaluation," *IEEE Trans. Software Eng.*, vol. 16, pp. 710–722, July 1988.

[5] J. R. Falcone, "A programmable interface language for heterogeneous distributed systems," *ACM Trans. Comput. Syst.*, vol. 5, pp. 331–351, 1987.

[6] A. P. Birrell and B. J. Nelson, "Implementing remote procedure calls," *ACM Trans. Comput. Syst.*, vol. 2, no. 1, pp. 39–59, Jan. 1984.

[7] Sun Microsystems, "Remote procedure call protocol specification version 2 (RFC 1057)," Network Inf. Center, SRI Int., June 1988.

[8] T. H. Dineen, P. J. Leach, N. W. Mishkin, J. N. Pato, and G. L. Wyant, "The network computing architecture and system: An environment for developing distributed applications, *Proc. Summer USENIX Conf.*, 1987, pp. 385–398.

[9] B. Liskov and L. Shrira, "Promises: Linguistic support for efficient asynchronous procedure calls in distributed system, *Proc. SIGPLAN'88 Conf. Programming Language Design and Implementation*, June 1988, pp. 260–267.

[10] E. F. Walker, R. Floyd, and P. Neves, "Asynchronous remote operation execution in distributed systems," *Proc. 10th Int. Conf. Distrib. Computing Syst.*, pp. 253–259, May 1990.

[11] A. L. Ananda, B. H. Tay, and E. K. Koh, "ASTRA—An asynchronous remote procedural call facility, *Proc. 11th Int. Conf. Distrib. Computing Syst.*, May 1991, pp. 172–179.

[12] K. S. Yap, P. Jalote, and S. Tripathi, "Fault tolerant remote procedure call," *Proc. 8th Int. Conf. Distrib. Computing Syst.*,, San Jose, CA, 1988, pp. 48–54.

[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*. Englewood Cliffs, NJ: Prentice-Hall, 1990.

[14] D. K. Gifford and N. Glasser, "Remote pipes and procedures for efficient distributed communication," *ACM Trans. Comput. Syst.*, vol. 6, pp. 258–283, Aug. 1988.

[15] B. Liskov and R. Scheifler, "Guardians and actions: Linguistic support for robust, distributed programs," *ACM Trans. Programming Languages Syst.*, vol. 5, pp. 381–404, July 1983.

[16] Y. Huang and C. V. Ravishankar, "Cicero: A protocol construction language," Tech. Rep. CSE–TR–171–93, Dept. of Elec. and Comput. Sci., Univ. of Michigan, Ann Arbor, 1993.

[17] C. V. Ravishankar and R. Finkel, "Linguistic support for dataflow," Tech. Rep. CSE–TR–14–89, Dept. of Elec. Eng. and Comput. Sci., Univ. of Michigan, Ann Arbor, 1989.

[18] ISO, *Information Processing Systems–Open System Interconnection—LOTOS—A Formal Description Technique Based on the Temporal Ordering of Observational Behavior*, ISO 8807 1988, 1985.

[19] ——, *Information Processing Systems—Open System Interconnection—Estelle (Formal Description Technique Based on an Extended State Transition Model)*, ISO9074 1988, 1987.

[20] Y. Huang and C. V. Ravishankar, "A universal RPC toolkit," Tech. Rep. CSE–TR–170–93, Dept. of Elec. Eng. and Comput. Sci., University of Michigan, Ann Arbor, Michigan, 1993.

[21] L. Peterson, N. Hutchinson, S. O'Malley, and H. Rao, "The x-kernel: A platform for accessing Internet resources," *IEEE Comput.*, vol. 23, pp. 23–33, May 1990.

[22] S. W. O'Malley and L. L. Peterson, "A dynamic network architecture," *ACM Trans. Comput. Syst.*, vol. 10, pp. 110–143, May 1992.

[23] J. M. Purtilo, "The polylith software bus," Tech. Rep. TR-2469, Comput. Sci. and Inst. for Advanced Comput. Studies, Univ. of Maryland, 1990.

[24] P. B. Gibbons, "A stub generator for multilanguage RPC in heterogeneous environments," *IEEE Trans. Software Eng.*, vol. 13, pp. 77–87, Jan. 1987.

[25] J. P. Ansart, P. D. Amer, V. Chari, J. F. Lenotre, L. Lumbroso, E. Mariani, and E. Mattera, "Software tools for Estelle," in *Protocol Specification, Testing and Verification VI (IFIP/WG 6. 1)*, B. Sarikaya and G. v. Bochmann, Eds. Amsterdam, The Netherlands: North-Holland, 1987.

[26] J. P. Briand, M. C. Fehri, L. Logrippo, and A. Obaid, "Executing LOTOS specifications," in*Protocol Specification, Testing and Verification VI (IFIP/WG 6.1)*, B. Sarikaya and G. v. Bochmann, Eds. Amsterdam, The Netherlands: North-Holland, 1987.

[27] S. T. Vuong, A. C. Lau, and R. I. Chan, "Semiautomatic implementation of protocols using an Estelle-C compiler," *IEEE Trans. Software Eng.*, vol. 14, pp. 384–393, Mar. 1988.

[28] D. P. Anderson, "Automated protocol implementation with RTAG," *IEEE Trans. Software Eng.*, vol. 14, pp. 291–300, Mar. 1988.

[29] L. Svobodova, "Implementing OSI systems," *IEEE J. Select. Areas Commun.*, vol. 7, pp. 1115–1130, Sept. 1989.

[30] C. Pu, H. Massalin, and J. Ioannidis, "The synthesis kernel," *Computing Syst.*, vol. 1, pp. 11–32, Winter 1988.

Y.-M. Huang received the B.S. degree in chemical engineering from National Taiwan University in 1982; dual M.S.E. degrees in chemical engineering and computer information and control engineering (CICE) from the University of Michigan, Ann Arbor, in 1986; and the Ph.D. degree in computer science and engineering from the University of Michigan, Ann Arbor, in 1993.

He is currently with IBM Corp., Research Triangle Park, NC. His current research interests include distributed systems and computer networks.

C. V. Ravishankar (S'82–M'84–S'85–M'85–S'86–M'86) received the B.Tech. degree in chemical engineering from the Indian Institute of Technology, Bombay, in 1975; and the M.S. and Ph.D. degrees in computer sciences from the University of Wisconsin, Madison, in 1986 and 1987, respectively.

He has been with the Department of Electrical Engineering and Computer Science at the University of Michigan, Ann Arbor, since 1986. His teaching and research at the University of Michigan have been in the areas of programming languages and distributed systems. His present research interests include large-scale distribution, heterogeneity, protocol synthesis, real-time systems, and database systems.

Dr. Ravishankar is a member of the IEEE Computer Society, ACM, and the Software Systems Research Laboratory and Real-Time Computing Laboratory at the University of Michigan.