Accommodating RPC Heterogeneities In Large Heterogeneous Distributed Environments[1]

Yen-Min Huang and Chinya V. Ravishankar
Dept. of EECS, The University of Michigan, Ann Arbor, MI. 48109
yenmin@eecs.umich.edu ravi@eecs.umich.edu

1 Introduction

Many RPC semantics have been designed and implemented in recent years to meet various application-specific requirements. Examples are synchronous RPC, asynchronous RPC, fault tolerant RPC, broadcast RPC, maybe RPC (no-return RPC), RPC with atomic transactions, and RPC with call-back mechanism [1, 2]. With emerging applications like multimedia conferencing and distributed real-time applications, it is conceivable that even more RPC protocols will be designed and implemented. This diversity of RPC protocols makes us adopt a general view of RPC as a protocol above OSI 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, greatly reducing the availability of software and resources in a large heterogeneous distributed environment. This problem has been addressed by HRPC/HCS [3] on a smaller scale, where the number of different RPC instances is small, RPC protocols are similar, and new RPC instances are rarely introduced. However, these characteristics do not always hold in a large heterogeneous environment. In such an environment, an acceptable solution must handle a large number of RPC instances, diversified RPC protocols, and rapid RPC protocol evolution at low software development and maintenance costs.

Our RPC agent synthesis scheme is a such solution [4]. Our system also includes a mechanism for distributing and acquiring synthesis information, and it supports RPC protocol evolution with minimum disturbance to the environment. Our approach is best for cross RPC within the same class of RPC semantics, for example, cross RPC among at-most-once RPCs, or among at-least-one RPCs. Although cross RPC between dissimilar RPC classes is also allowed, it is not fully explored in the current work because there may exist substantial mismatches between the client and the server code, and it may require significant modification to client/server code to resolve differences. Since it is impossible for us to determine user-level semantics intended, it is the user's responsibility to make sure that the client and the server programs can be meanfully interconnected, and to provide the appropriate agent synthesis specifications to interconnect them. This paper is focused on the design and implementation of an agent synthesis scheme.

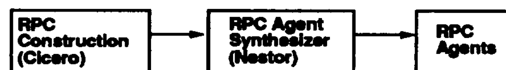

Figure 1: The RPC Agent Synthesis Scheme

Broadly speaking, our RPC agent synthesis scheme has two components (see Figure 1): a set of language constructs (Cicero) to describe RPC protocol constructions, and a program (Nestor) to synthesize and activate RPC agents automatically. A novel feature of Cicero is the use of *event patterns* [5] to control synchrony, asynchrony and concurrency in protocol execution (Section 4.1). Nestor is a remote evaluation system [6, 7] specialized for synthesizing RPC agents. The major differences between Nestor and other remote evaluation systems are that Nestor uses a different language (Cicero) for evaluation (agent synthesis), and this evaluation usually happens at the client site instead of the server site.

2 Agent Synthesis Scheme Design

2.1 Determining Agent Configurations

There are two possible choices for agent configuration: the one-agent configuration and the two-agent configuration (Figure 2). The one-agent configuration consists of a gateway agent which interconnects two

[1]This work was partly supported by the Consortium for International Earth Sciences Information Networking.

0-8186-1060-3425/93 $03.00 © 1993 IEEE

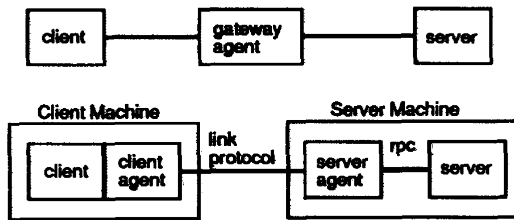

Figure 2: One-Agent and Two-Agent Configurations

programs using different RPC protocols. The two-agent configuration can be constructed by splitting the gateway agent into two agents connected by a link protocol. These two agents, the client and the server agent, are placed on the client and the server machines respectively. These agents may be linked into client or server programs depending whenever the respective program may be modified.

The two-agent configuration is used for our synthesis scheme because it results in a much cleaner synthesis scheme than the one-agent configuration. It is cleaner because each agent only needs to know the the local and the link RPC protocols, and the runtime support for both the native and the link RPC protocols are locally available. However, the one-agent configuration will not have all the runtime support available locally because the single gateway agent could be located on either the client or server machine. In either case, it must be aware of the details of the RPC not available locally. This makes the agent construction and synthesis more difficult. Synthesizing a gateway agent on a third-party machine is not considered here, because of access control constraints.

2.2 Handling RPC Heterogeneities

Two RPCs may differ in their call semantics, failure semantics, RPC topology, external data representation, and so on. Dealing with these heterogeneities all together is not easy even using a synthesis scheme.

To make our synthesis scheme manageable, RPC heterogeneities are handled differently depending upon their types. RPC heterogeneities are classified into those that are ***semantics-dependent***, and those which are ***semantics-independent***. For example, heterogeneities in call semantics and failure semantics are semantics-dependent, while heterogeneities in RPC message format are semantics-independent because they are an artifact of implementation and have no effect on RPC semantics. Semantics-dependent heterogeneities are handled by synthesizing the implementation of the specified semantics directly, giving programmers maximum flexibility in describing their RPC semantics implementation to achieve better cross-RPC performance. Semantics-independent heterogeneities are handled by providing a default implementation of each mechanism, which is encapsulated in the link protocol and has little effect on applications. Thus, call semantics, failure semantics, and RPC topology, must all be described in the protocol construction language and synthesized by the synthesizer.

3 Nestor: An Agent Synthesizer

Nestor is a runtime organization for RPC agent synthesis. Its synthesis support consists of a set of libraries and utility programs used by Nestor to synthesize RPC agents. The libraries provide the functions to implement the link protocol between two agents. The utility programs are used to generate and package code to form an agent. Specifically, the utility programs consist of compilers for Cicero and C, a stub generator, and a software packaging utility (UNIX ***make***).

To synthesize an RPC agent, the synthesis support uses three specifications: 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. The agent profile specification determines how an agent will be synthesized and managed. For each protocol, two sets of specifications are needed: one for the client agent and one for the server agent.

Figure 3 illustrates how utility programs use these specifications 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 implementing the link protocol. The stub generator compiles the RPC interface specification and generates the stub routine which interfaces with the client or the server program. When an agent must contact its client/server program using native RPC, the native stub generator is used to generate the stub for constructing the agent. If a stub generator is not available, users are required to provide RPC stubs. Finally, the RPC stub, libraries, and the link-protocol implementation are linked together to form an agent. This entire synthesis process is specified in a file (***makefile***), whose location is obtained from an RPC agent profile specification.

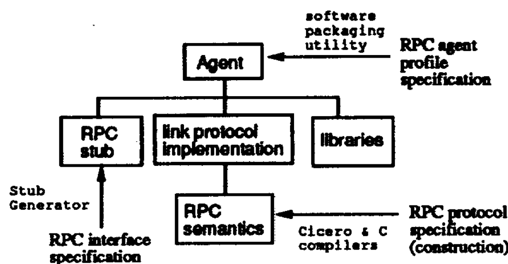

Figure 3: Specifications and Components of an Agent

4 Cicero: Language Constructs for Protocol Construction

Cicero is a set of language constructs designed to add multi-threaded control capability to an existing programming language to make protocol construction easier. It is used in our system to describe the implementation of RPC protocol semantics (Section 2.2).

Although Cicero constructs may be applied to different programming languages, we have chosen C as our target language because it has efficient compilers and is portable. Therefore, our prototype implementation of Cicero includes a compiler for translating Cicero constructs into C code and a Cicero runtime library providing implementation of these constructs. The **when** construct is the most important in Cicero. It represents the execution control mechanism as well as the unit of parallelism in Cicero.

The central notion the **when** construct is based on is *event patterns*, which provide a dataflow execution model. The dataflow model is chosen because it represents maximal parallelism, and is used in today's high-performance architecture/microprocessors. Also, it can be translated into Petri nets [8] to take advantage of existing protocol verification methods/tools [9].

4.1 Event Patterns

Each *event* is a unbounded sequence of its event instances. An *event instance* is an object representing an occurrence of an event. For example, if timeouts are modeled as events, then the third occurrence of timeout is represented by the third timeout event instance. While instances of an event may occur at several places in a program, all instances of an event are globally ordered and delivered in order.

Because complex dependencies often exist among event instances, Cicero borrows a feature called *event patterns* from the language Post [5] to help programmers express such dependencies. An event pattern specifies the precise relationships between event instances that trigger actions in a protocol, and it is used by programmers to control synchrony, asynchrony, and sequencing in protocol construction. We recognize three basic relationships between two event instances: synchrony, asynchrony, or sequentiality. In Cicero, these relationships are expressed by operators called *event combinators* which may be used to combine simpler event patterns into more complex ones. The three event combinators are "$\wedge$", ",", and "$\sim$" corresponding to synchrony, asynchrony, and sequencing, respectively. Their semantics are formally defined as Petri nets (Figure 4) and are described as follows.

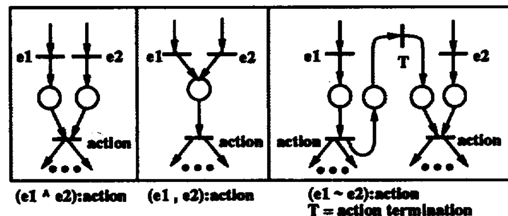

Figure 4: The Petri Net Definition for Event Patterns

1. **when** ($e1 \wedge e2$): *actions* **end**
   This program requires that the associated actions be triggered only when both $e1$ and $e2$ occur.

2. **when** ($e1$, $e2$): *actions* **end**
   This pattern requires that when either $e1$ or $e2$ occurs the associated actions be triggered, and the action instances triggered by $e1$ and $e2$ are executed concurrently.

3. **when** ($e1 \sim e2$): *actions* **end**
   This pattern requires the actions triggered by $e1$ and $e2$ to be executed in sequence. No actions may be triggered by $e2$ unless all actions triggered by $e1$ are finished.

5 Implementation and Performance

Our prototype version of Nestor and Cicero runs on top of BSD 4.3 and Mach 2.5. The Cicero compiler is implemented using UNIX *lex* and *yacc*, and its runtime library has interfaces to existing thread packages such as SUN LWP, and Mach Cthreads.

The performance of synthesized agents depends on the overhead of Cicero runtime and communication primitives. The Cicero runtime overhead comprises the overhead of managing event instances and threads. Depending on the thread package used, the runtime overhead of using event patterns varies. On average, it accounts for less than 10% of the overall RPC latency. To measure the overhead of communication primitives, we constructed AT1-RPC protocols (at-most-once RPC semantics) using the same RPC semantics and transport mechanism as Sun RPC. For typical RPC messages ($\leq$ 1K), our performance is competitive with SUN RPC's performance (ours is 3% slower on average).

It is harder to get good performance on cross RPCs, because the extra indirection involved makes it inherently slower than homogeneous RPC. Also, slow native RPC performance can easily become a performance bottleneck when the client programs involved may not be modified. Our measurements are taken for cross RPCs among SUN RPC, Mach RPC and HP/Apollo NCS/NCA RPC using a two-agent configuration. The latency for cross RPCs is about 1.5 to 2.5 times longer than sending messages through TCP directly. The worst case occurs when the cross RPC is performed on a lightly-loaded local Ethernet with a slow local native RPC on the server side. However, the absolute performance for this case (9.7ms for cross RPC and 3.6ms for TCP) is still tolerable in most situations.

6 Conclusions

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. The agent synthesis scheme provides a solution with low software development and maintenance costs. This feature is particularly advantageous for the client site, since a client can contact many servers using different RPC protocols without having implementations of each server protocols.

Importing protocol constructions from the outside also offers other advantages. It provides immediate software availability after a protocol construction is created or updated. It minimizes disturbance when updating existing RPCs and introducing new RPCs. Hence, RPC protocol evolution is well supported. It offers the opportunity to synthesize specialized code to improve performance. Finally, it also makes the synthesis solution scalable, and makes each site fully autonomous.

A synthesis scheme must be backed up by a good specification language. Cicero represents a significant part of this effort. It provides the following advantages under an acceptable overhead: (1) support for multiple-thread execution (**when** constructs) for exploiting parallelism, (2) event patterns to control synchrony, asynchrony, and sequentiality in protocol execution, which is a better implementation paradigm than using just thread package alone, (3) coarse-grain parallelism by combining smaller event patterns into larger ones, (4) translation to Petri net to be able to take advantage of existing protocol verification methods/tools.

References

[1] B. H. Tay and A. L. Ananda. A Survey of Remote Procedure Calls. *Operating Systems Review*, 24(3):68–78, July 1990.

[2] A. L. Ananda, B. H. Tay, and E. K. Koh. A Survey of Asynchronous Remote Procedure Calls. *Operating Systems Review*, 26(2):92–109, April 1992.

[3] 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 Transactions on Software Engineering*, 13(8):880–894, August 1987.

[4] Y. Huang and C. V. Ravishankar. Accommodating RPC Heterogeneities Using Automatic Agent Synthesis. Technical Report CSE-TR-131-92, Dept. of EECS, The University of Michigan, Ann Arbor, Michigan, 1992.

[5] C. V Ravishankar and R. Finkel. Linguistic Support for Dataflow. Technical Report CSE-TR-14-89, Dept. of EECS, The University of Michigan, Ann Arbor, Michigan, 1989.

[6] J. R. Falcone. A Programmable Interface Language for Heterogeneous Distributed Systems. *ACM Transactions on Computer Systems*, 5(4):331–351, 1987.

[7] J. W. Stamos and D. E. Gifford. Implementing Remote Evaluation. *IEEE Transactions on Software Engineering*, 16(7):710–722, July 1988.

[8] K. M. Kavi, B. P. Buckles, and U. N. Bhat. Isomorphism Between Petri nets and Dataflow Graphs. *IEEE Transactions on Software Engineering*, 13(10):1127–1134, October 1987.

[9] T. Suzuki, S. M. Shatz, and T. Murata. A Protocol Modeling and Verification Approach Based On A Specification Language and Petri Nets. *IEEE Transactions on Software Engineering*, 16(5):523–536, May 1990.