Distrib. Syst. Engng **4** (1997) 191–202. Printed in the UK PII: S0967-1846(97)82909-9

Secure synthesis and activation of protocol translation agents

Yen-Min Huang and Chinya V Ravishankar†

Department of Electrical Engineering and Computer Science, The University of Michigan, Ann Arbor, MI 48109-2122, USA

Received 29 July 1996

Abstract. Protocol heterogeneity is pervasive and is a major obstacle to effective integration of services in large systems. However, standardization is not a complete answer. Standardized protocols must be general to prevent a proliferation of standards, and can therefore become complex and inefficient. Specialized protocols can be simple and efficient, since they can ignore situations that are precluded by application characteristics.

One solution is to maintain agents for translating between protocols. However, n protocol types would require $O(n^2)$ agents, since an agent must exist for a source–destination pair. A better solution is to create agents as needed.

This paper examines the issues in the creation and management of protocol translation agents. We focus on the design of Nestor, an environment for synthesizing and managing RPC protocol translation agents. We provide rationale for the translation mechanism and the synthesis environment, with specific emphasis on the security issues arising in Nestor. Nestor has been implemented and manages heterogeneous RPC agents generated using the Cicero protocol construction language and the URPC toolkit.

1. Introduction

Heterogeneity is an inevitable concomitant of distribution. It complicates interaction between entities in a distributed system, and is a major obstacle to effective integration of services in large systems. Protocol heterogeneity often arises when systems are developed in isolation, but a more important cause is that protocols are specialized to meet application-specific requirements. Standardization is not always a good answer, since standardized protocols must target the general case in a problem domain to prevent a proliferation of standards. Such general protocols can be complex and inefficient. A strong case is made for application-specific protocols in [3, 4]. Such protocols can be simple and efficient, since they can safely ignore conditions precluded by application characteristics, but which more general protocols must address.

A good example of this phenomenon is the domain of remote-procedure calls. Figure 1, which was modified from [5] to include synchronous RPC, illustrates various RPC protocols available today. It is significant that these protocols all differ significantly in their semantics, not simply in their mechanics. Despite this great variety, there will almost certainly be a continued need to build RPCs with new semantics, because it is unlikely that current RPCs address the requirements of all future applications. Further, RPCs customized to application semantics will improve throughput, response time, and failure resilience even for existing distributed applications. As distributed applications become more sophisticated, customized RPCs can also reduce the effort of application development if they include more functionalities common to a specific class of applications. For example, it is easier to implement distributed transaction applications by using RPC with atomic transactions than by using traditional RPCs. Within emerging areas such as multimedia conferencing and distributed real-time applications, it is likely that more RPC protocols will be designed and implemented.

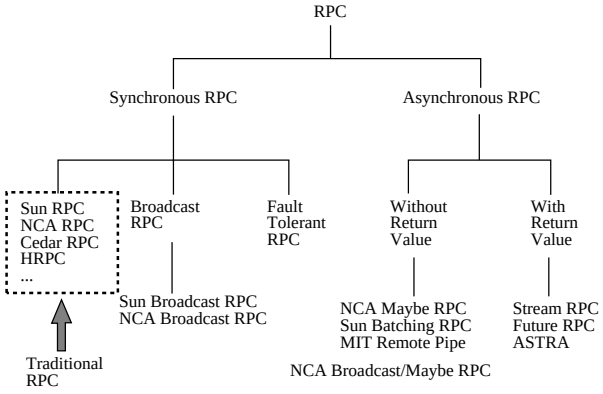


Figure 1. The current RPC domain.

Standardization of protocols and interfaces is a common approach to handing the problem of heterogeneity. Good

† E-mail address: ravi@eecs.umich.edu

examples of such standards are CORBA's general inter-ORB protocol (GIOP) and internet inter-ORB protocol (IIOP) [6]. However, this approach may not always work, particularly for legacy software, since it is unrealistic to expect that all existing software will be recoded to new standards. Another problem is that standards must address the general case in a problem domain, for otherwise, a separate standard will be required for each specific situation or context, leading to a proliferation of standards. Such generality rules out application- or situation-specific customization, though communication patterns between applications can be quite specialized. For example, [7] observes that over 95% of remote-procedure calls do not cross machine boundaries, and proposes a customized, light-weight RPC machanism which is over three times as fast as native RPC. Also, a general-purpose protocol must be robust in the face of arbitrary program behaviour, while a protocol customized to an application can ignore behaviours that the programmer or compiler knows will never occur.

Another solution to handling the problem of protocol heterogeneity is to maintain agents for translating between protocols. However, an agent must exist for a source–destination pair in such a solution, requiring a total of $n * (n - 1)$ agents for n different protocol types. This is a clearly unacceptable rate of growth in the number of agents. CORBA's approach [6] is to provide for environment-specific inter-ORB protocols (ESIOPs) to handle cases where certain distributed computing infrastructures may already be in use. An example is the DCE common inter-ORB protocol (DCE-CIOP) which is useful when existing applications use DCE. This approach combines standardization with the use of predefined cross-domain mappings. Thus, it appears not to scale well. Clearly, a better solution would be to create agents as needed.

1.1. Addressing heterogeneity

This paper concentrates on the issues that arise in creating and executing protocol-translation agents at client sites in a secure fashion. However, to provide background, we outline the issues that motivate the agent synthesis approach. Some of this exposition also appears in [1] and [3], which provide details on how the agent synthesis process operates in the case of RPC heterogeneities.

The RPC domain is a good model of the more general issues of protocol heterogenity, and we use it to illustrate more general issues. As figure 1 shows, different RPC protocols are often designed to satisfy different application requirements, making it hard for programs built using different RPC protocols to be interconnected directly. This difficulty greatly reduces the availability of software and resources in a large heterogeneously distributed environment, and increases the costs of developing and maintaining distributed applications with multiprotocol support. This problem is most acute when trying to integrate distributed applications from different sources. Some examples are integrating different distributed file systems, implementing federated distributed databases, or grouping various scientific computation servers. The same problem also arises when building an application with multiple-protocol support, such as a client that can communicate with different name servers using different RPCs, or a server that must accept requests from many clients.

We encountered this problem whilst working on the *client–service* model in the Cygnus distributed system [8]. The classical client–server model requires the client both to identify servers as well as to speak the server protocol. Therefore, clients are typically able to interact only with a limited number of servers in such systems. The Cygnus system [8] solves this difficulty by introducing the *client–service* model, in which clients request *services*, not servers. The mapping from services to servers is performed by Cygnus. Although the work on Cygnus was done independently of CORBA (and in fact, largely predates it), their respective approaches have both similarities and differences. Cygnus and CORBA both present the client with an object-oriented programming model which emphasizes transparency and hides many of the system details from the client, and in which object operations may execute either locally or remotely. However, Cygnus completely eliminates the notion of server, so that objects represent *service*. Thus, when a service provider (server) fails, Cygnus is often able to reconfigure the connection to another equivalent server, and continue the identical service. Cygnus also assumes that servers are completely insular, so it may be required to operate in an environment with little standardization.

Since clients do not see servers at all, the Cygnus system must handle all protocol heterogeneities. Since servers in Cygnus are insular, clients must conform to server protocols. Cygnus uses agent synthesis as its solution to this difficulty. When a remote server must be contacted to obtain service, a protocol specification is retrieved from the server site, and an agent is synthesized at the client site to map to the server protocol. In CORBA, standards exist for communications within and across ORBs [6]. Work is already in progress [9, 10] on building effective implementations of cross-ORB communications using bridges and half-bridges, particularly for DCE environments. Rosenberry and Teague [11] discussed the interesting issue of the interoperability of DCE and windows NT domains, the two environments likely to be among the most pervasive in the future. However, in contrast to CORBA and other such efforts, Cygnus does not have the notion of ORBs. Any client–server connection established as a result of a service request may be a connection between entities speaking different protocols. Cygnus synthesizes code to handle such heterogeneity on demand. We have a full implementation for handling heterogeneous remote-procedure calls, and we observe no loss of performance over native RPC (see section 3.4). We have built both customized RPCs with specialized semantics (multicast, at-most-once, call-back, asynchronous, etc) and cross-RPC (Sun, Apollo, Mach, NCS, etc) implementations in our experiments. For further details and examples see [1–3].

Another real issue with RPCs is that the cost of implementing a new RPC system is usually much higher than it needs to be. Much development effort is often

spent on redoing significant parts of supporting facilities, such as stub generators, name servers, and low-level communication functions. The main obstacle to reusing these components is that they are often tightly integrated with the RPC runtime, and modifications are often required to support new RPC features. Therefore, without clever design of the RPC runtime interface and architecture, it would be impossible to provide highly reusable RPC runtime components useful for a variety of RPC protocols.

1.2. Our approach: agent synthesis

An agent synthesis scheme uses a synthesizer to generate implementations of RPC agents from high-level descriptions, and is attractive because it meets our requirements well, and since much of the effort of coding agents can be saved. Also, there are fewer restrictions on the kinds of RPC agents that can be described and generated. Therefore, if designed properly, a synthesis scheme can provide a more general solution than existing approaches [12–16], and with much lower agent development and maintenance costs.

There is a major difference between our agent synthesis scheme and that of others [17, 18]: in addition to traditional stubs, we also generate implementations of RPC protocol machines as a part of the RPC runtime. In other words, for each different RPC protocol, a different RPC runtime may be generated along with the necessary stubs. This synthesis capability can also be used for rapid prototyping of new RPC systems [3], which is very useful to RPC developers.

Our system provides two services: *customization* and *cross-RPC service*. Customization supports fast prototyping of customized RPC systems and for experimenting with new RPC features and semantics. This service is designed to help prototype a new RPC system, including the RPC runtime, the stub generator, and the name server. Cross-RPC service supports cross-RPC communication to increase software availability. This service assumes a common transport-layer protocol between the client and the server machines, and provides an interconnecting program that preserves the largest subset of RPC semantics common to the two RPCs.

It is economical to provide these two services in one system because cross-RPC service can be built on top of customized RPC service. 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. RPC agents are gateways that can translate between different RPC protocols. Therefore, implementing RPC protocols is the central task for building RPC agents, and the customized RPC service is designed for exactly this type of task.

Broadly speaking, our RPC agent synthesis scheme (see figure 2) has two components: a language to describe RPC protocol constructions, and a runtime environment to synthesize and activate RPC agents automatically. Agents are synthesized by combining libraries with the implementation code generated from the specifications.

Our agent synthesis scheme is realized through three subsystems: a universal RPC toolkit (URPC toolkit) [3], a protocol construction language (Cicero) [2], and a runtime support system for agent synthesis and management (Nestor). The URPC toolkit provides semantics-independent library functions, allowing programmers to prototype diversified new RPC systems with minimal coding effort. Cicero is an executable specification language designed for describing complex protocol constructions. Nestor is the subsystem integrating the URPC toolkit, Cicero, and other software utilities to perform agent synthesis and management for cross-RPC communication. To facilitate agent synthesis, Nestor also provides support for importing protocol constructions from remote hosts. This paper focuses on Nestor.

2. The mechanics of agent synthesis

This section provides an introduction to the mechanics of synthesizing agents for RPC protocols using Nestor. Some of this discussion appears in [1], and is reproduced here to provide a background for our discussion. The agent synthesis system comprises a specification language Cicero [2], the URPC toolkit for generating RPC runtimes [3], and Nestor, the environment in which the synthesis and activation of agents takes place. This paper focuses on Nestor.

Nestor facilitates the synthesis of agents with various RPC semantics. It provides services for importing and exporting specifications, and can activate and terminate agents for cross-RPC communication. Therefore, Nestor is responsible for the entire lifecycle of RPC agents, i.e. their creation, activation and termination.

2.1. 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 the components and libraries to create executable images of agents. The first step is accomplished by programmers manually, and the second step is accomplished by Nestor with little or no intervention from programmers.

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 specification together determine what agent will be synthesized. Specifically, the RPC protocol construction (written in Cicero) describes the implementation of RPC semantics. The RPC interface specification describes the RPC operation interface, and is used to generate stubs to interface with the client, the server and the URPC runtime library. The agent profile specification determines how an agent will be synthesized and managed. It contains instructions for synthesizing and managing agents. For example, the agent profile specification defines the synthesis environment, and the 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

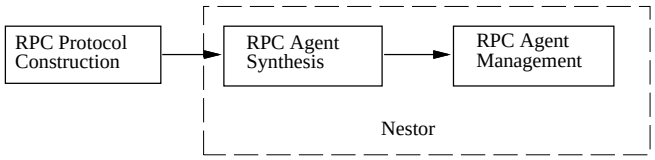


Figure 2. The RPC agent synthesis scheme.

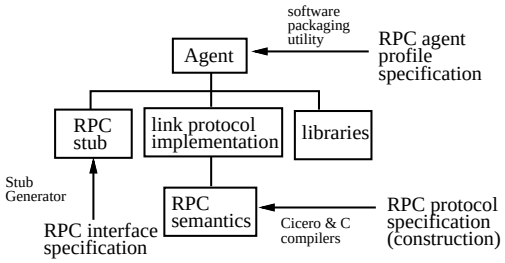


Figure 3. Specifications and components of an agent.

agent. The remote interface specification and RPC protocol construction language have already been described in [3] and [2] respectively, and RPC agent profile specification will be described in section 2.3.

Nestor uses a set of libraries and utility programs to synthesize executable images of RPC agents. These libraries include the URPC and Cicero runtime libraries. The URPC runtime library provides routines for constructing communication mechanisms between agents, and the Cicero runtime library supports Cicero language constructions for describing the link protocol of agents. The utility programs used by Nestor consist of compilers for Cicero and C, stub generators, and a software packaging utility (e.g. the UNIX *make*).

Figure 3 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 which interface with the client or server program, and with the link protocol implementation. For customized RPC service, Nestor uses the URPC stub generator to generate stubs to link with client and server programs. For cross-RPC service, in addition to the URPC stub generator, 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.

2.2. 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 since 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 user can import the client–agent construction from the server. The transfer support is introduced to encourage sharing 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. Users would just import the new specifications and synthesize local agents. It minimizes disturbance when updating existing RPCs and introducing new RPCs. 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. For details on how protocols may be specified and used, see [1–3].

The agent-management support is responsible for controlling the activation, execution and the termination of agents. All 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 can be used to activate the client and the server directly.

2.3. RPC agent profile specification

An RPC agent profile specification consists of a list of attributes and values, defining how agents can be synthesized, activated and when they are to be terminated. The attributes currently supported are listed below, and are categorized according to their usage. Figure 4 shows an example RPC profile specification.

- **Identity related attributes**
  - *sp_title*: This attribute defines the name of an RPC agent profile specification, and can be referred by programmers. This name, independent of the specification file name, is used to match up client and server agent profile specifications.

sptitle	= test
role	= client
instance_id	= 0
spec_server	= taipei.eecs.umich.edu
work.dir	= /users/yenmin/test
spec_path	= /users/yenmin/test
import_list	= test.rif test_fsm.cic
export_list	=
makecmd	= make test_cl
exec_cmd	= test_cl
time_to_live	= 300
acls	= 1540 2008
end	

Figure 4. Example RPC profile specification.

  - *role*: This attribute indicates the type of the specification, i.e. whether it is for a client agent or a server agent.

- **Environment related attributes**
  - *spec_server*: This attribute contains the address of the host, which will participate in transferring specifications. The direction of the transfer will be given by the user at the time of transfer.
  - *work_dir*: This attribute indicates the working directory where the agent will be synthesized and activated.
  - *spec_path*: This attribute indicates the directory where the specifications are kept locally, so that Nestor can know the source and destination for the specification transfer.
- **Transfer-support related attributes**
  - *import_list*: This attribute lists the files that must be imported before synthesizing a local agent.
  - *export_list*: This attribute lists the files that can be exported when synthesizing a remote agent. The *import_list* and *export_list* are provided to support automatic transferring a list of specifications.
- **Synthesis and management related attributes**
  - *make_cmd*: This attribute contains the command for synthesizing the agent.
  - *exec_cmd*: This attribute indicates the command for executing the agent.
  - *time_to_live*: This attribute defines the lifetime of the agent.
- **Security related attributes**
  - *acls*: This attribute contains the ACL for users who are allowed to synthesize or activate the agent from a given specification.

3. 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 3.3 for details).

3.1. Step 1: specification construction

The client and server programs are built on top of SUN RPC and HP/Apollo NCS RPC respectively. To keep the example simple, we assume that there is only one operation (`printmsg()`) exported by the server, which displays a given string. This operation is defined in NCS interface definition language (NIDL) as follows

```
[uuid(4448ecb46000.0d.00.00.fe.da.00.00.00),
     port(dds:[19], ip:[6677]), version(1)]

interface printmsg_intf

{ void printmsg( handle_t      [in]  h,
                 string0[1024] [in]  msg,
                 int           [out] *result);
}
```

To make the example more interesting, let us assume that the client cannot be modified and must access this operation through an existing SUN RPC interface, which is defined in SUN RPC interface definition language as follows

```
program MESSAGEPROG {
   version MESSAGEVERS {
      int PRINTMESSAGE(string) = 1;
   } = 1;
} = 99;
```

We can see that although the client and the server can be interconnected logically, their RPC protocols and RPC interfaces are different. To perform cross RPC, these heterogeneities must be resolved. In general, these will not be the only kinds of heterogeneities encountered. For example, there may also be differences in protocol semantics. For details on how these are resolved see [1–3]. Our primary interest here is the mechanics of agent synthesis and activation.

To focus on the synthesis mechanism, let us assume that the client has already obtained the RPC protocol construction and RPC interface specification for the server. These two specifications are required when a server is exported to the outside world, and may be stored as part of a ‘name service’ database that covers a local domain. The RPC interface specification is defined by using our interface definition language, as follows

```
[ aptitle = printmsg; version = 1.0; ]
{ int urpc_printmsg( [in]        handle_t  h,
                     [in,string] char      *msg,
                     [out]       int       *result);
}
```

Although our interface definition language has a different syntax, one can readily see that the above RPC interface specification can be easily derived from the server’s remote interface definition.

Figure 5 illustrates the structure of the client agent. The client agent consists of five parts: a SUN RPC server stub, a link-protocol client stub, a piece of connector code, the SUN-RPC runtime library, and the URPC runtime library. The SUN RPC server stub is generated from the given interface definition by the SUN RPC stub generator. The link protocol client stub is generated by the URPC stub generator from the RPC interface specification. The connector code is introduced to bind the server agent, resolve mismatches between interfaces, and to glue the native stub and link-protocol stub together. For the client

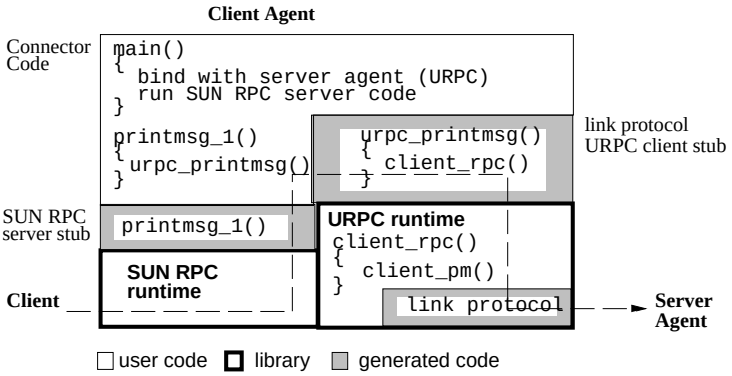


Figure 5. A synthesized cross-RPC client agent.

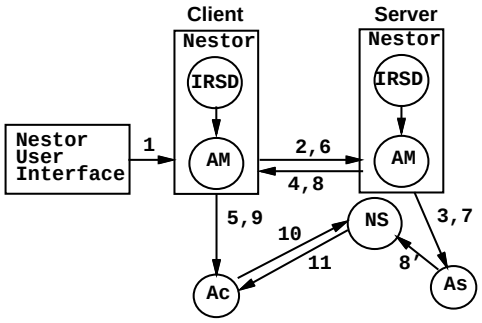


Figure 6. The agent synthesis process in Nestor.

agent, the connector code consists of a main program for establishing bindings and a set of dummy server operations which we will subsequently call the entry routines in the link-protocol URPC client stub. The SUN RPC and URPC runtime libraries are assumed to be available on the client host. However, it is possible that an implementation of the client protocol machine is not readily available. In this case, the implementation of the client protocol machine must either be generated from an imported protocol construction or coded by programmers. Finally, to produce the client agent, Nestor will instruct the package utility to compile and link all these parts together. Due to symmetry, the server agent can be similarly synthesized.

3.2. Step 2: agent synthesis and activation

After all the specifications are in place, the programmer instructs Nestor to synthesize agents. The steps in the agent synthesis and activation are shown as numbered arcs in figure 6.

The Nestor runtime 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 IRSDs. Upon receiving a request, IRSD forks off a copy of the 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 user interface (NUI). It allows the user to interact with Nestor by issuing commands.

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 3–5). After the agents are synthesized, the client AM notifies the server AM to activate the synthesized server agent (steps 6 and 7). After the server agent (As) is activated, the client agent (Ac) is activated, and the port number used by the server agent is looked up by the client agent through the server-side URPC name server (steps 8–11). Now, the agents are ready to perform the specified heterogeneous RPC.

3.3. Name service support

Name service is not the focus of this work; 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 using a global database† which has knowledge of all available services in the network. The port number is obtained through the server-side URPC name server.

Nestor uses the URPC name service as its default RPC name-service mechanism to bypass name-service heterogeneity problems. 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 using

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

Table 1. Comparison of URPC-ATM1/UDP with SUN-RPC/UDP.

Data size	URPC ATM1/UDP		SUN-RPC/UDP	
	Local	LAN	Local	LAN
null	2.68 ms	2.54 ms	2.68 ms	2.68 ms
1 K	3.59 ms	4.37 ms	3.74 ms	4.51 ms
2 K	5.08 ms	6.37 ms	5.86 ms	7.21 ms

Table 2. Comparison of URPC-ATM1/TCP with SUN-RPC/TCP.

Data size	URPC ATM1/TCP		SUN-RPC/TCP	
	Local	LAN	Local	LAN
null	3.24 ms	3.19 ms	3.96 ms	3.64 ms
1 K	3.98 ms	5.02 ms	5.33 ms	5.55 ms
2 K	5.55 ms	6.61 ms	7.14 ms	7.36 ms

the native RPC runtime 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.

3.4. Performance

A detailed discussion of the performance of our agent synthesis system appears in [1, 3]. Here we reproduce some relevant numbers. Our data indicate that the performance of our synthesized code is usually as good as that of native implementations, and often better. There are two reasons for this somewhat non-intuitive result. First, customization can increase the semantic content of individual messages, and thus reduce the number of messages required. The benefit gained from customization obviously depends on the application and the manner in which the customization is performed, but can be significant. A second benefit can be a reduction in time for individual calls since customization results in implementations tailored for specific situations, which do not incur the overhead present in implementations that address more general situations. To facilitate such performance comparisons, we constructed an RPC (URPC-ATM1) with at-most-once failure semantics using a protocol machine implementation similar to that of SUN RPC. Comparisons of elapsed time were made between URPC-ATM1 and SUN RPC on different transport-layer protocols (UDP and TCP), and the results are listed in tables 1 and 2. For a more detailed analysis of these results, see [1, 3].

4. Security support

Security support is necessary because Nestor operates in a large heterogeneous distributed environment containing many different administrative domains. Any point-to-point communication in this environment may require messages to travel through many different administrative domains.

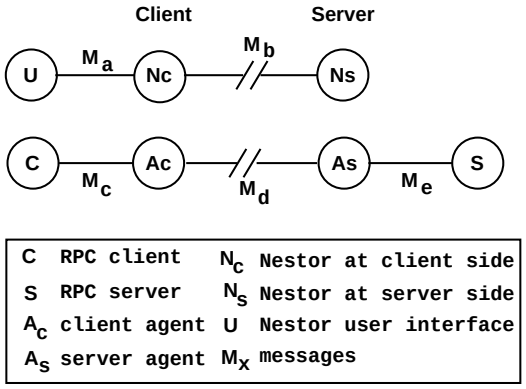


Figure 7. Objects in Nestor.

Without proper security mechanisms, Nestor users and service providers are subject to various security attacks. To provide reasonable protection to Nestor users and service providers, the following security issues must be addressed:

- *maintaining the integrity of specifications*: The RPC protocol specifications are sent over the network, so the integrity of the specifications must be preserved. If a specification in transit were to be modified, the consequences would be potentially very serious.
- *Controlling access to Nestor services*: We may want to regulate the permission to synthesize agents from any given specification.
- *Providing secrecy for RPC communication*: The integrity of messages between agents must be guaranteed.

Although these issues are typical in a distributed environment, the design of the security mechanism for Nestor requires special attention in two areas:

- The security mechanism must handle the delegation relationships between the client/server and their agents, and between the client and client/server Nestor instances. This delegation relationship requires authentication through intermediaries, where traditional client–server authentication protocols [19–22] cannot be readily applied.
- The security mechanism must detect the impersonation of agent executable images, since agents may be synthesized well in advance of their use.

The first step in designing a security scheme is to examine the objects participating in Nestor activities which may be subject to security attacks. These objects are shown in figure 7, and can be processes, executable images, or messages. Each process and its executable image are represented by a circle in figure 7; messages passed between two processes are denoted by the symbol M subscripted with a label identifying the connection between the two processes (e.g. M_a). Our mechanism does not prevent an intruder from exploiting existing security flaws in a system by becoming a superuser [23]. This type of attack is a general problem for all systems, not just for Nestor.

Our mechanism is based on the Needham–Schroeder public-key protocols [19] with three assumptions. The first assumption is that public keys are distributed by trusted name servers whose public keys are known by Nestor. The second assumption is that public keys of each client and Nestor instance are already advertised among name servers.

Table 3. Protection for messages.

Message	Protected by
M_a	digital signature
M_b	digital signature
M_d	session key encryption
M_c/M_e	native RPC system and kernel

Table 4. Detection of impersonated objects.

Object	Impersonation detection scheme
Client (U/C)	digital signature
Client Nestor (Nc)	digital signature
Server Nestor (Ns)	digital signature
Client agent (Ac)	session key encryption/decryption
Server agent (As)	session key encryption/decryption

The third assumption is that the private keys of the user and Nestor instances involved are not compromised.

4.1. Protecting messages

One of our major concerns is protecting the integrity and secrecy of messages exchanged between machines. We assume that the local interprocess communication is *secure*. In other words, the integrity and the secrecy of local messages are guaranteed by the operating system delivering them. The security mechanism design will focus on protecting messages exchanged between machines.

Two types of intermachine messages need to be protected: messages passed between the client and server Nestor instances (M_b) and messages passed between agents (M_d). The schemes used for message protection are summarized in table 3.

Messages passed between the client and server Nestor instances are important because they may contain specifications and keys, whose unforgeability and accountability are important to both the client and the server. To deal with intermediaries, we adopt the cascading authentication protocol proposed in [24] to protect these messages. The messages exchanged between Nestor instances are digitally signed [19] by the user and each Nestor instance, so that their integrity is protected, and the client and each Nestor instance are identifiable. Therefore, under the cascading authentication protocol, the messages M_a must also be digitally signed even though they are passed locally. This security protocol for exchanging messages between Nestor instances is discussed in section 4.5.

The messages passed between agents (M_d) are protected using shared session key encryption/decryption instead of the public-key encryption. Shared key encryption is used because keys can be created and distributed safely by Nestor instances, and shared key encryption is about 1000–5000 times faster than public-key encryption [22]. A new session key is created and passed to the client and the server agents each time they are invoked. This shared session key is also used to implement a challenge-response protocol, which client and server agents use to authenticate each other. This security protocol for protecting messages is discussed in section 4.6.

4.2. Access control for Nestor services

We use ACLs to control the access to Nestor services. The server-side Nestor instance authenticates a client by decrypting the client's request which is digitally signed by the user and the client-side Nestor instance†. Because this client authentication is a direct result of the digital-signature protection scheme used for messages, no additional mechanism is necessary. Note that the Nestor ACL is not responsible for controlling access to servers. It only controls access to Nestor services.

4.3. Protection against impersonated executable objects

Protection against impersonation should consist of two parts: substitution prevention and substitution detection. Substitution prevention for executable objects depends mostly upon setting access rights carefully and removing security flaws, which are both functions of system administrators. Thus, here we focus upon substitution detection.

Substitution detection involves detecting two forms of substitution: process impersonation and executable image substitution. Our substitution detection mechanism uses either public-key or session-key encryption to detect impersonation of processes, and uses a *digest function* [25] to detect the impersonation of executable images (see table 4). The same digital signature scheme used in protecting messages is also used for detecting the impersonation of the user (U), the RPC client (C) and Nestor processes (N_c and N_s). When they fail to identify themselves through encrypted messages, the receiving process detects possible impersonation and closes the connection.

The shared session key is used to detect the impersonation of agent processes. Since it is possible to impersonate an agent after the agent is activated but before the client contacts it‡, the agent must be authenticated when contacted by the client. This authentication between the client and the Ac is accomplished through a typical challenge-response protocol using a shared session key encryption/decryption (see figure 9). However, the distribution of the session key requires participation of the client program, which may be difficult. In such cases, the impersonation may not be detected.

To ensure that the synthesized agent is indeed the one that is activated, Nestor can use a digest function to compute the digest§ of the binary image of an agent when

† A request is first encrypted using the user's private key, and passed to the local Nestor instance. The local Nestor forwards the encrypted request to the server-side Nestor by further encrypting the message using its own private key and the server-side Nestor's public key. The server-side Nestor can decrypt the message using its private key, the client-side Nestor's public key, and user's public key.

‡ This is the time that the intruder can register a malicious agent in the native name service to replace the genuine one.

§ The digest is the result of a one-way digest function. In other words, it is computationally impossible to compute the input from a given digest (the output).

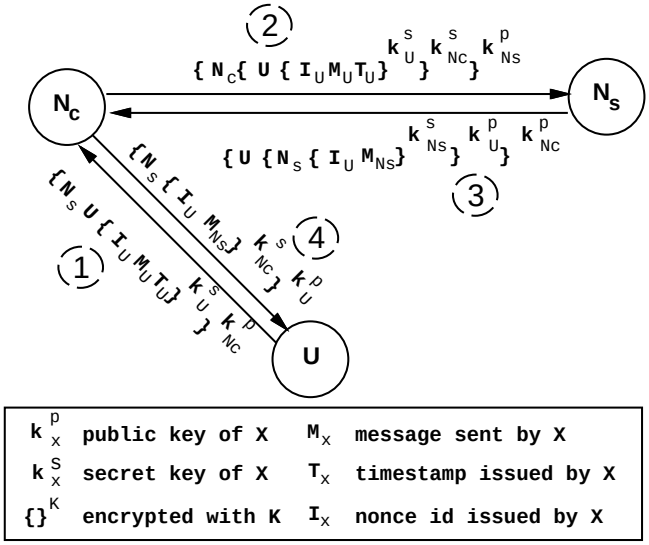


Figure 8. Security protocol for passing messages between Nestor instances.

the agent is synthesized. Before an agent is executed, the digest of its binary image is computed and compared with the digest computed when the agent was synthesized.

4.4. Cross-RPC authentication

The native RPC runtime in a cross-RPC agent enables the agent to communicate with the client or server using native secure communication protocols. Although different encryption/decryption schemes are subsumed by the native runtime in the agent, mapping principal identities across administration domains remains an issue.

This identity-mapping issue can be addressed differently depending on whether or not client and server RPCs use a common authentication protocol. If client and server RPCs use a common authentication protocol, such as Kerberos [21], then the client principal can register directly in the server domain as a foreign entity, and can be authenticated through Kerberos security servers. Similarly, the Nestor instances can use Kerberos for authentication with each other and with the corresponding client or server. After mutual authentication, the client and all agents will be supplied with a shared session key by Kerberos security servers for secure communication.

If there are no common identities, Nestor relies on a public key encryption scheme for authenticating the client and Nestor instances. The authenticated client's identity must be mapped into an equivalent principal on the server side, which is the preselected identity for the server agent. This mapping may or may not be possible. Assuming an appropriate identity can be found, a session key shared between the client and the agents can be generated and distributed by the client-side Nestor. This session-key distribution protocol is described and formally analysed in section 4.5.

4.5. Security protocol between Nestor instances

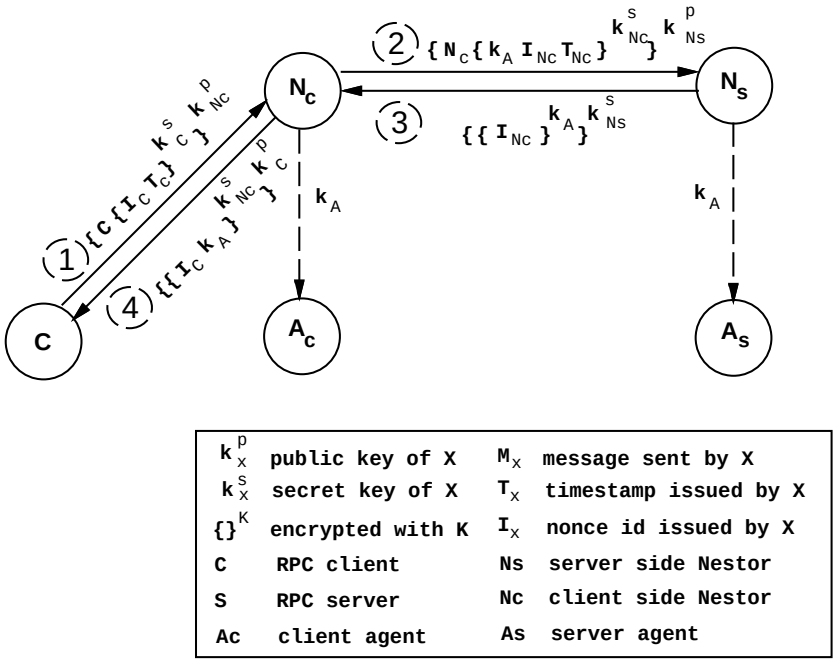
The security protocol for exchanging messages between Nestor instances is illustrated in figure 8. Timestamps and nonce numbers are introduced to guard against the replay attack and process impersonation (see section 4.3).

The security protocol between Nestor instances works as follows. When a user wishes to request a service, the Nestor user interface encrypts the user's message with the user's private key and the client-side Nestor's public key before sending to the client-side Nestor. Upon receiving the user message, the client-side Nestor decrypts the message using its own private key, and then re-encrypts the user message with its private key and the server-side Nestor's public key before forwarding the message to the server-side Nestor. The server-side Nestor can decrypt this message with its own private key, the client-side Nestor's public key, and user's public key. After the user's request is granted and processed, the reply message is returned, and the reply message is encrypted with the user's public key, the client-side Nestor's public key, and the server-side Nestor's private key. The client-side Nestor unwraps (decrypts) the message with its private key, and passes the resulting message to the Nestor user interface. At the end, the Nestor user interface extracts the content of the reply message by decrypting the message with his private key and the server-side Nestor's public key. A formal analysis of this protocol can be found in section 5.1.

4.6. Security protocol between agents

Messages passed between agents are protected using shared session key encryption/decryption. A new session key is created and passed to the client and the server agents each time they are invoked. This shared session key is also used to implement a challenge-response protocol, which client and server agents use to authenticate each other.

The security protocol between agents is basically a session key distribution protocol. The session key is generated by the client-side Nestor upon client request. The session key is distributed using digitally signed messages to the client and the server-side Nestor. It is also passed to



k^p_X	public key of X	M_X	message sent by X
k^s_X	secret key of X	T_X	timestamp issued by X
$\{\}^K$	encrypted with K	I_X	nonce id issued by X
C	RPC client	Ns	server side Nestor
S	RPC server	Nc	client side Nestor
Ac	client agent	As	server agent

Figure 9. Security protocol for passing messages between agents.

client and server agents as one of the activation parameters. This distribution protocol is illustrated in figure 9. This protocol also uses timestamps to guard against replay attack and nonce numbers as shared secrets for process impersonation detection (see section 4.3). Once the session key distribution is completed, agents can use this session key to encrypt/decrypt messages. In section 5.2 there is a formal analysis of this protocol.

5. Security protocol analysis

To analyse Nestor security protocols, the logic proposed in [26] is used. All the analysis steps are also illustrated in the same notation and style as in [26]. The rules in figure 10 also appear in [26] and will be used for analysing our security protocols.

5.1. Security protocol between nestor instances

5.1.1. Messages.

M1. $U \rightarrow N_c : \{N_s, U, \{I_u, M_u, T_u\}_{K_u^s}\}_{K_{N_c}^p}$
M2. $N_c \rightarrow N_s : \{N_c\{U\{I_u, M_u, T_u\}_{K_u^s}\}_{K_{N_c}^s}\}_{K_{N_s}^p}$
M3. $N_s \rightarrow N_c : \{U\{N_s\{I_u, M_{N_s}\}_{K_{N_s}^s}\}_{K_u^p}\}_{K_{N_c}^p}$
M4. $N_c \rightarrow U : \{N_s\{I_u, M_{N_s}\}_{K_{N_s}^s}\}_{K_u^p}$

5.1.2. Idealized protocol translation.

M1. $U \rightarrow N_c : \{\{I_u, M_u, T_u\}_{K_u^s}\}_{K_{N_c}^p}$
M2. $N_c \rightarrow N_s : \{\{I_u, M_u, T_u\}_{K_u^s}\}_{K_{N_c}^s}\}_{K_{N_s}^p}$
M3. $N_s \rightarrow N_c : \{\{\{I_u, M_{N_s}\}_{K_{N_s}^s}\}_{K_u^p}\}_{K_{N_c}^p}$
M4. $N_c \rightarrow U : \{\{I_u, M_{N_s}\}_{K_{N_s}^s}\}_{K_u^p}$

R1 : $\dfrac{P \text{ believes } Q \overset{K}{\longleftrightarrow} P,\ P \text{ sees } \{X\}_K}{P \text{ believes } Q \text{ said } X}$

R2 : $\dfrac{P \text{ believes } \overset{K^p}{\longmapsto} Q,\ P \text{ sees } \{X\}_{K^s}}{P \text{ believes } Q \text{ said } X}$

R3 : $\dfrac{P \text{ believes } Q \overset{Y}{\rightleftharpoons} P,\ P \text{ sees } \{X\}_Y}{P \text{ believes } Q \text{ said } X}$

R4 : $\dfrac{P \text{ believes fresh } (X),\ P \text{ believes } Q \text{ said } X}{P \text{ believes } Q \text{ believes } X}$

R5 : $\dfrac{P \text{ believes } Q \text{ controls } X,\ P \text{ believes } Q \text{ believes } X}{P \text{ believes } X}$

R6 : $\dfrac{P \text{ believes fresh } (X)}{P \text{ believes fresh } (X,Y)}$

R7 : $\dfrac{P \text{ believes } Q \overset{K}{\longleftrightarrow} P,\ P \text{ sees } \{X\}_K}{P \text{ sees } X}$

R8 : $\dfrac{P \text{ believes } \overset{K^p}{\longmapsto} P,\ P \text{ sees } \{X\}_{K^p}}{P \text{ sees } X}$

R9 : $\dfrac{P \text{ believes } \overset{K^p}{\longmapsto} Q,\ P \text{ sees } \{X\}_{K^p}}{P \text{ sees } X}$

R10 : $\dfrac{P \text{ sees } (X,Y)}{P \text{ sees } X}$

Figure 10. Analysis rules.

5.1.3. Goals.

G1 : N_s believes U said M_u
G2 : N_s believes fresh(M_u)
G3 : N_s believes U believes M_u
G4 : U believes N_s said M_{N_s}
G5 : U believes fresh(M_{N_s})
G6 : U believes N_s believes M_{N_s}

5.1.4. Assumptions.

A1 : U believes $\overset{K^p_{N_c}}{\longmapsto} N_c$
A2 : U believes $\overset{K^p_{N_s}}{\longmapsto} N_s$
A3 : U believes M_u
A4 : U believes I_u
A5 : U believes fresh(I_u)
A6 : N_c believes $\overset{K^p_{N_s}}{\longmapsto} N_s$
A7 : U believes $\overset{K^p_u}{\longmapsto} U$
A8 : N_s believes $\overset{K^p_u}{\longmapsto} U$
A9 : N_s believes $\overset{K^p_{N_c}}{\longmapsto} N_c$
A10 : N_s believes M_{N_s}
A11 : N_s believes fresh(M_{N_s})
A12 : N_s believes fresh(T_u)

5.1.5. Protocol analysis.

M1, R8 :	N_c sees $\{I_u, M_u, T_u\}_{K^s_u}$	E1
M2, R8 :	N_s sees $\{\{I_u, M_u, T_u\}_{K^s_u}\}_{K^s_{N_c}}$	E2
E2, A9, R2 :	N_s believes N_c said $\{I_u, M_u, T_u\}_{K^s_u}$	E3
A8, R2 :	N_s believes U said $\{I_u, M_u, T_u\}$	E4
R10 :	N_s believes U said M_u	(G1)
R12, R6 :	N_s believes fresh(I_u, M_u, T_u)	E5
E5 :	N_s believes fresh(M_u)	(G2)
G1, G2, R4 :	N_s believes U believes M_u	(G3)
M3, R8 :	N_c sees $\{\{I_u, M_{N_s}\}_{K^s_{N_s}}\}_{K^p_u}$	E6
M4, R8 :	U sees $\{I_u, M_{N_s}\}_{K^s_{N_s}}$	E7
E7, A2, R2 :	U believes N_s said $\{I_u, M_{N_s}\}$	E8
E8 :	U believes N_s said M_{N_s}	(G4)
E7, A4, R6 :	U believes fresh(M_{N_s})	(G5)
G4, G5, R4 :	U believes N_s believes M_{N_s}	(G6)

5.2. Security protocol between agents

5.2.1. Messages.

M1. $C \rightarrow N_c : \{C, \{I_c, T_c\}_{K^s_c}\}_{K^p_{N_c}}$
M2. $N_c \rightarrow N_s : \{N_c\{K_a, I_{N_c}, T_{N_c}\}_{K^s_{N_c}}\}_{K^p_{N_s}}$
M3. $N_s \rightarrow N_c : \{\{I_{N_c}\}_{K_a}\}_{K^s_{N_s}}$
M4. $N_c \rightarrow C : \{\{I_c, K_a\}_{K^s_{N_c}}\}_{K^p_u}$

5.2.2. Idealized protocol translation.

M1. $C \rightarrow N_c : \{\{I_c, T_c\}_{K^s_c}\}_{K^p_{N_c}}$
M2. $N_c \rightarrow N_s : \{\{A_c \overset{K_a}{\longleftrightarrow} A_s, I_{N_c}, T_{N_c}\}_{K^s_{N_c}}\}_{K^p_{N_s}}$
M3. $N_s \rightarrow N_c : \{I_{N_c}, A_c \overset{K_a}{\longleftrightarrow} A_s\}_{K^s_{N_s}}$
M4. $N_c \rightarrow C : \{\{I_c, C \overset{K_a}{\longleftrightarrow} A_c\}_{K^s_{N_c}}\}_{K^p_u}$

5.2.3. Goals.

G1 : A_c believes $A_c \overset{K_a}{\longleftrightarrow} A_s$
G2 : A_s believes $A_c \overset{K_a}{\longleftrightarrow} A_s$
G3 : C believes $C \overset{K_a}{\longleftrightarrow} A_c$
G4 : N_s believes $A_c \overset{K_a}{\longleftrightarrow} A_s$
G5 : N_c believes N_s believes $A_c \overset{K_a}{\longleftrightarrow} A_s$

5.2.4. Assumptions.

A1 : C believes $\overset{K^p_{N_c}}{\longmapsto} N_c$
A2 : C believes fresh(I_c)
A3 : C believes N_c controls $C \overset{K_a}{\longleftrightarrow} A_c$
A4 : N_c believes $\overset{K^p_{N_s}}{\longmapsto} N_s$
A5 : N_c believes $\overset{K^p_c}{\longmapsto} C$
A6 : N_c controls $A_c \overset{K_a}{\longleftrightarrow} A_s$
A7 : N_c controls $C \overset{K_a}{\longleftrightarrow} A_c$
A8 : N_c believes fresh(T_c)
A9 : N_c believes fresh(I_{N_c})
A10 : N_s believes $\overset{K^p_{N_c}}{\longmapsto} N_c$
A11 : N_s believes fresh(T_{N_c})
A12 : N_s believes N_c controls $A_c \overset{K_a}{\longleftrightarrow} A_s$
A13 : A_c believes N_c controls $A_c \overset{K_a}{\longleftrightarrow} A_s$
A14 : A_c believes fresh($A_c \overset{K_a}{\longleftrightarrow} A_s$)
A15 : A_c believes N_c said $A_c \overset{K_a}{\longleftrightarrow} A_s$
A16 : A_s believes N_s controls $A_c \overset{K_a}{\longleftrightarrow} A_s$
A17 : A_s believes fresh($A_c \overset{K_a}{\longleftrightarrow} A_s$)
A18 : A_s believes N_s said $A_c \overset{K_a}{\longleftrightarrow} A_s$

5.2.5. Protocol analysis.

A14, A15, R4 :	A_c believes N_c believes $A_c \overset{K_a}{\longleftrightarrow} A_s$	E1
E1, A13, R5 :	A_c believes $A_c \overset{K_a}{\longleftrightarrow} A_s$	(G1)
A17, A18, R4 :	A_s believes $N_s m$ believes $A_c \overset{K_a}{\longleftrightarrow} A_s$	E2
E2, A16, R5 :	A_s believes $A_c \overset{K_a}{\longleftrightarrow} A_s$	(G2)
M4, R8 :	C sees $\{I_c, C \overset{K_a}{\longleftrightarrow} A_c\}_{K^s_{N_c}}$	E3
E3, R2 :	C believes N_c said $\{I_c, C \overset{K_a}{\longleftrightarrow} A_c\}$	E4
A2, R6 :	C believes fresh($C \overset{K_a}{\longleftrightarrow} A_c$)	E5
E4, E5, R4 :	C believes N_c believes $C \overset{K_a}{\longleftrightarrow} A_c$	E6
E6, A3, R5 :	C believes $C \overset{K_a}{\longleftrightarrow} A_c$	(G3)
M2, R8 :	N_s sees $\{A_c \overset{K_a}{\longleftrightarrow} A_s, I_{N_c}, T_{N_c}\}_{K^s_{N_c}}$	E7
E7, R2 :	N_s sees N_c said $\{A_c \overset{K_a}{\longleftrightarrow} A_s, I_{N_c}, T_{N_c}\}$	E8
E8, A11, R6 :	N_s believes fresh($A_c \overset{K_a}{\longleftrightarrow} A_s$)	E9
E8, E9, R4 :	N_s believes N_c believes $A_c \overset{K_a}{\longleftrightarrow} A_s$	E10
E10, A12, R5 :	N_s believes $A_c \overset{K_a}{\longleftrightarrow} A_s$	(G4)
M3, R2 :	N_c believes N_s said $\{I_{N_c}, A_c \overset{K_a}{\longleftrightarrow} A_s\}$	E11

A9, R6 : N_c believes
fresh($A_c \overset{K_a}{\longleftrightarrow} A_s$) E12

E11, E12, R4 : N_c believes N_s
believes $A_c \overset{K_a}{\longleftrightarrow} A_s$ (G5)

6. Conclusions

Automating the generation of protocol agents can be very useful in the presence of heterogeneity. This paper presents a method for automating the process of agent synthesis and activation. To illustrate our scheme, we provide details for creating and managing remote-procedure calls agents, since RPCs are the most widely used paradigm for interprocess communication. However, the techniques described in this paper appear to extend to other paradigms as well. Since security is of a particular concern in distributed environments, our scheme incorporates security protocols to guard against various kinds of attacks.

The agent synthesis mechanisms in Nestor are currently operational, and are being used in conjunction with the RPC agent synthesis mechanism described in [1]. We are currently completing the implementation of the security mechanisms.

Acknowledgments

This work was partly supported by NASA and its Socioeconomic Data and Applications Center operated by the Consortium for International Earth Sciences Information Networking.

References

[1] Huang Y and Ravishankar C V 1994 Designing an agent synthesis system for cross RPC communication *IEEE Trans. Software Engng* **20**

[2] Yen-Min Huang and Ravishankar C V 1994 Linguistic support for controlling protocol execution *Proc. 14th Int. Conf. on Distributed Computing Systems* pp 581–8

[3] Huang Y and Ravishankar C V 1996 URPC: A universal RPC toolkit *Comput. J.* **39**

[4] Felten E 1992 The case for application-specific communication protocols *Proc. Intel Supercomputer Systems Division Technology Focus Conf.* pp 171–81

[5] Ananda A L, Tay B H and Koh E K 1992 A survey of asynchronous remote procedure calls *Operating Syst. Rev.* **26** 92–109

[6] Vinoski S 1997 CORBA: Integrating diverse applications within distributed heterogeneous environments *IEEE Commun. Mag.* **14**

[7] Bershad B N, Anderson T E, Lazwoska E D and Levy H M 1990 Lightweight remote procedure call *ACM Trans. Comput. Syst.* **8** 37–55

[8] Rong Chang and Ravishankar C V 1994 A service acquisition mechanism for server-based heterogeneous systems *IEEE Trans. Parallel Distrib. Syst.* **5**

[9] Zhonghua Yang and Vogel A 1996 achieving interoperability between CORBA and DCE applications using bridges *Proc. IFIP/IEEE Int. Conf. on Distributed Platforms* (London: Chapman and Hall) pp 144–55

[10] Steinder M, Usok A and Zieliński K 1996 A framework for inter-ORB request level bridge construction *Proc. IFIP/IEEE Int. Conf. on Distributed Platforms* (London: Chapman and Hall) pp 86–99

[11] Rosenberry W and Teague J 1993 *Distributing Applications Across DCE and Windows NT* (Sebastapol, CA: O'Reilly and Associates)

[12] Bershad B N, Ching D T, Lazowska E D, Sanislo J and Schwartz M 1987 A remote procedure call facility for interconnecting heterogeneous computer systems *IEEE Trans. Software Engng* **13** 880–94

[13] Auerbach J 1990 TACT: A protocol conversion toolkit *IEEE J. Selected Areas Commun.* **8** 143–59

[14] Sun Microsystems 1991 *Open Network Computing—RPC Programming*

[15] Stamos J W and Gifford D E 1988 Implementing remote evaluation *IEEE Trans. Software Engng* **16** 710–22

[16] Falcone J R 1987 A programmable interface language for heterogeneous distributed systems *ACM Trans. Comput. Syst.* **5** 331–51

[17] Purtilo J M 1990 The Polylith software bus *Technical Report TR-2469* Computer Science and Institute for Advanced Computer Studies, University of Maryland

[18] Gibbons P B 1987 A stub generator for multilanguage RPC in heterogeneous environments *IEEE Trans. Software Engng* **13** 77–87

[19] Needham R M and Schroeder M D 1978 Using encryption for authentication large networks of computers *Commun. ACM* **21** 993–9

[20] Birrel A D 1985 Secure communication using remote procedure call *ACM Trans. Comput. Syst.* **3** 1–14

[21] Steiner J G, Neuman B C and Schiller J I 1988 Kerberos—An authentication service for open network systems *Proc USENIX Winter Conf.* pp 191–202

[22] Lampson B, Abadi M, Burrows M and Wobber E 1991 Authentication in distributed systems: theory and practice *ACM Operating Syst. Rev.* **25** 165–82

[23] Grampp F T and Morris R H 1984 Unix operating system security *Technical Report 8* AT&T Bell Laboratories Technical Journal

[24] Sollins K R 1988 Cascaded authentication *IEEE Symp. on Security and Privacy* pp 156–63

[25] Rivest R 1990 The MD4 message digest algorithm *Technical Report TM 434* Laboratory of Computer Science MIT

[26] Burrows M and Abadi M 1990 A logic of authentication *ACM Trans. Comput. Syst.* **8**