# URPC: A Toolkit for Prototyping Remote Procedure Calls

YEN-MIN HUANG AND CHINYA V. RAVISHANKAR

*Electrical Engineering and Computer Science Department, The University of Michigan, Ann Arbor, MI 48109-2122, USA*
*Email: ravi@eecs.umich.edu*

**Many new remote procedure calls (RPC) systems are being built to meet different application requirements, and much development effort has been spent on redoing significant parts of the RPC system. This paper describes URPC, a toolkit for prototyping new RPC systems. It allows programmers to provide high-level implementations of RPC semantics and to customize supporting RPC services, such as stub generation and name service, to match the requirements of different RPC semantics. This approach increases flexibility in constructing new RPC systems and greatly reduces coding effort. In addition, this approach allows application-specific optimization by increasing the semantic content of individual RPC calls through customization, as well as by allowing programmers to import protocol machine implementations. Thus, the generated prototype RPC implementations can perform as fast as native RPCs.**

*Received January 30, 1996; revised June 26, 1996*

## 1. INTRODUCTION

A variety of remote procedure calls (RPC) semantics has been designed and implemented in recent years. Some examples are synchronous RPC, asynchronous RPC, fault-tolerant RPC, multicast RPC and RPC with atomic transactions [1, 2]. One reason for the continued need to design RPCs with new semantics is simply that existing RPCs cannot possibly address all the requirements of future applications. However, as we argue in Subsection 1.1, there are other important reasons for customizing the semantics of communication protocols to exploit application semantics. Customization can make RPCs more lightweight and simplify implementation. It can also make it possible for a single client to use RPCs with differerent semantics simultaneously. This feature will be especially important since we expect much functionality that presently resides at the user level in many application classes to be factored out and be moved into lower levels. Such downward migration is a primary reason for the increasing diversity of RPC semantics.

Existing distributed applications will also benefit from new RPC mechanisms if they improve throughput, response time and failure resilience. As distributed applications become more sophisticated, new RPCs will also be useful in reducing applications development effort by incorporating more features common to a specific class of applications. For example, it is easier to implement distributed transaction applications using RPC with atomic transactions than with traditional RPCs. It is also likely that more RPC protocols will be designed and implemented to support emerging areas like multimedia conferencing and distributed real-time applications.

Most innovation in a new RPC system resides in its protocol machine implementation. However, other components like stub generators and name servers must also be implemented to interface with new RPC runtimes. Even within the protocol machine implementation, a large portion of the development effort is usually spent in managing internal data structures and implementing low-level details. We have found [3] that the entire protocol machine implementation comprises only 30–40% of the code of an RPC system, and high-level protocol code accounts for even less. As a result, much development effort is being spent on the less critical parts of the RPC system.

This paper describes URPC, an RPC toolkit for rapidly prototyping RPC runtimes. This toolkit focuses on providing quick implementations of special-purpose RPC semantics, and is not intended for generating very general, DCE-like RPC implementations. It targets RPC designers, not application programmers, and may also have particular utility as a research tool.

### 1.1. Standardization versus customization

Standardization is one approach used to deal with the issue of diversity. Standardization clearly has its place, and particularly benefits future applications development. However, the approach also has its drawbacks. First, it does not solve the interoperability problem for existing applications, since it is unrealistic to recode them all to new standards. In such cases, generating cross-RPC implementations [3] remains the only option. Even worse, standards usually address the general case in a problem domain to avoid a proliferation of standards. That often rules out application- or situation-specific customization.

The advantages of such customization are becoming

increasingly apparent. A persuasive case has already been made for such customization in communication protocols [4, 5]. Communication patterns can be quite specialized, and the penalty for using the general solution for special cases can be severe. For instance, [6] observes that over 95% of RPC calls do not cross machine boundaries, and proposes a customized, light-weight RPC mechanism, whose performance is better than that of SUN RPC by a factor of over three. A complementary example of customized, intra-application, but cross-machine RPC is Rx, the RPC mechanism used by the Andrew File System [7]. This RPC mechanism views a single RPC call as the serial instantiation of two byte streams, corresponding to the call and the response. It also permits arguments of arbitrary size, and allows either the client or server to stream calls actively.

Thus, customization is not restricted to the level of implementation, and can be carried to the level of semantics. This argument is made forcefully by Felten [4], who argues that optimum performance of a message-passing protocol requires that the protocol be designed to exploit the semantics of individual applications. He argues, using examples, that a general-purpose protocol must be robust in the face of arbitrary program behavior, but an application-specific protocol can ignore certain behaviors that the programmer or compiler knows will not occur.

Thus, the issues of generality and robustness become more manageable if the environment in which the RPC protocol will be used is more restricted, as is the case for customized RPCs. Some other examples of the benefits of the flexibility of customized RPCs could be the ability to return specific information pertaining to remote failure modes, the ability to handle aborts from the client end, and the ability to place application-specific recovery mechanisms into the protocol. Users may also want to change buffering behavior, alter timeout values or introduce multiple timeouts in designing RPC protocols with more complex semantics.

A simplified implementation is not the only advantage of customizing RPC semantics. It can also have performance benefits. Consider an application that submits transaction-like operations through RPC to a server that supports both interactive and batch modes. In the batch mode, transaction operations may be sent and processed in a batch, and the application can examine the results at its convenience. This submission process involves three steps: setting the transaction mode, issuing a series of operations, and indicating the end of this series. These steps can be modelled using the following RPC calls:

1. handle = `START` (mode, server);
2. `EXEC` (handle, trans_id, opcode, len, data);
3. `FINISH` (handle);
4. `RESULT` (handle, trans_id, &result_len, &result_data, &ret);

`START` is intially called, followed by a series of $n$ calls to `EXEC`, followed by a `FINISH` call. To obtain the results of such a series of operations, the application issues a `RESULT` call. To implement this using conventional RPC would require $n + 2$ RPC calls ($n$ `EXEC` calls, a `START` and a `FINISH`), causing $(n + 2) * 2$ message exchanges between client and server, including request and acknowledgement messages for $n + 2$ RPCs.

However, let us assume that the application semantics are such that intermediate results are not used by the application. In this case, there is no advantage in remotely executing each `EXEC` call as it is issued. If the programmer can control the buffer size and buffering policy in the RPC runtime, all transactions can be buffered and flushed to the server when `FINISH` is issued. The message count drops to 2 if the lower layer does not fragment this message, a significant performance improvement over the previous approach.

Another difficulty for conventional RPC in this example is that `EXEC` and `RESULT` may specify different RPC semantics, but they share a common binding (represented by the handle). RPC semantics are difficult to change on the fly. For example, perhaps `EXEC` uses may-be RPC, while `RESULT` uses synchronous RPC. Heterogeneous RPC semantics are also encountered when trying to integrate services in a large distributed system [8]. A single client may need to communicate with different servers running on different platforms. Such complications can be tricky to handle using conventional RPC.

It may sometimes be possible to simulate application-specific RPC semantics with a general package, say with DCE RPC. However, such a simulation would use a more heavy-weight system to simulate lighter-weight semantics, and is likely to be awkward and expensive. Our focus is on allowing users to customize RPCs with finer-grained primitives than provided by such general RPC packages.

### 1.2. Toolkit features

This toolkit is not intended for generating very general, DCE-like RPC implementations. Rather, it focuses on providing quick implementations of special-purpose RPC semantics, and may have particular utility as a research tool. Thus, this toolkit is presently intended for RPC designers, not for application programmers. The work in [6] described a special-purpose kernel-level RPC. In [5], the authors show that protocol implementations can be moved into user space to gain flexibility without sacrificing performance or security. Our work attempts to customize user-level RPC and describes a mechanism to achieve such customization.

URPC simplifies the task of RPC developers since they need provide only high-level protocol implementations. Thus, it works in concert with Cicero, a protocol description language. Other supporting components, like stub generators and name servers, are designed to be customizable to meet diversified RPC requirements,

thus greatly reducing the effort spent on integration. The ability to prototype RPC systems rapidly reduces development costs, and also allows RPC developers to explore a larger design space to produce better RPC protocol implementations.

In addition to fast prototyping, our toolkit provides other benefits, which have not been well supported in the past:

- It allows application developers to customize or replace RPC semantics to meet special application requirements. The URPC toolkit allows programmers to provide high-level protocol implementations to change RPC semantics.
- It allows application developers to fine tune RPC protocol performance for different environments.
- It provides a stable basis for evolution of RPC technology: the toolkit remains unchanged while new RPC protocols are imported.
- It provides a common paradigm for multiple dissimilar RPC semantics to coexist, allowing application developers to use different RPC semantics within an application. This is possible because the URPC toolkit can function as a common basis for implementing dissimilar RPC semantics.

Surprisingly, our toolkit can offer these features without sacrificing any application or RPC performance. In fact, an RPC system constructed from our toolkit often has better performance than the corresponding native RPC system. This paradox arises because existing RPC systems are optimized for the general case, and usually cannot be customized to take advantage of individual application characteristics and environment. Also, by allowing developers to specify a protocol machine implementation of an RPC, we preserve most of the optimization in handcrafted implementations. Thus, the resulting RPC can be just as fast as native RPC.

Our toolkit consists of a compiler for Cicero, our protocol construction language, a stub generator, a name server, and an RPC runtime library. An interesting aspect of our toolkit is the separation of mechanisms that are not dependent on RPC semantics from those that are. This separation allows a new RPC runtime to be created from a protocol description written either in C or in our protocol construction language Cicero [9]. This approach provides increased flexibility in constructing new RPC semantics and reducing coding effort.

Cicero is a language veneer designed to facilitate complex RPC protocol implementation. A novel feature of Cicero is the use of event patterns [10] to control synchrony, asynchrony and concurrency in protocol execution. To match the flexibility provided by the toolkit, our stub generator can be instructed to generate customized stub routines, incorporating various protocol implementations. Our name server uses a generic naming scheme, which can also be customized to meet different naming requirements. The RPC runtime library implements a generic RPC runtime architecture, and provides interfaces to communication primitives, name services, data conversion functions and customization functions.

In this paper, we will focus on the design of our toolkit, and illustrate the usage of the toolkit by constructing different RPC protocols. The rest of this paper is organized as follows. Section 2 discusses the design of the toolkit. Section 3 describes examples to illustrate how to use the toolkit to construct various RPC protocols. Section 4 introduces our protocol construction language Cicero. Section 5 compares the performance between the SUN RPC and an RPC constructed using our toolkit. Section 6 describes related work, and Section 7 concludes this paper.

## 2. TOOLKIT DESIGN AND ARCHITECTURE

Our toolkit design involved two considerations: (1) determining which parts of an RPC runtime should be specified by programmers and which should be provided by the toolkit libraries, and (2) designing the proper architecture and supporting facilities.

To maximize flexibility in constructing new RPC systems and to minimize coding effort, we must be selective about which RPC features should be specified by programmers. RPC features can be numerous and can vary greatly between different RPC systems. Examples of RPC features are call semantics, failure semantics, RPC topology, external data representation, naming and binding mechanism, the security mechanism, and so on.

Our approach is to first classify RPC features into those that are semantics-dependent and those that are semantics-independent, depending on their effect and importance to the application. We see features related to call semantics and failure semantics as semantics-dependent, and those relating to RPC message format as semantics-independent, since they are artifacts of implementation.

The toolkit supports RPC features differently depending upon their types. Default implementations are available for semantics-dependent features, but the toolkit also lets programmers provide their own implementations. In this way, the toolkit provides maximal flexibility in the implementation of RPC semantics, which comprise call semantics (synchronous or asynchronous RPC), failure semantics (e.g. at-most-once) and RPC topology (e.g. 1-to-1, 1-to-$N$, etc.).

For the semantics-independent parts, where the exact implementation is less crucial, our toolkit provides default implementations that programmers may customize to reduce coding effort. For example, programmers may care little what external data representation is used, as long as the toolkit provides one for data marshalling/unmarshalling. However, the toolkit does provide a mechanism for programmers to bypass the marshalling/unmarshalling mechanism if necessary,

so that programmers may choose between different conversion schemes like no conversion, conversion on one side, or conversion on both sides. Therefore, all mechanisms pertaining to semantics-independent features are provided through libraries or as runtime services. RPC feature classification makes the toolkit simpler and also greatly reduces the complexity of using the toolkit, with little or no detriment to the generality of the solution.

### 2.1. Toolkit architecture

The toolkit has overall architecture similar to that of common RPC systems, consisting of a runtime library, a stub generator and a name server. However, these components have been designed differently to provide the flexibility required by the toolkit. They are designed to be independent of the specific models or semantics that are associated with traditional RPC components. Programmers customize them for implementing different models or semantics.

#### *2.1.1. Runtime library*

The runtime library is linked with the application and stub routines to perform RPC. An interesting aspect of our runtime library is that it can import implementations of RPC semantics, and can be customized for different situations at runtime. To support these features, we selectively expose, through well-defined interfaces, parts of the URPC internal runtime architecture traditionally hidden from programmers.

To import RPC semantics implementations, the runtime library allows programmers to provide three routines, which together implement protocol machines in an RPC system (see shaded boxes in Figure 1). The client and server protocol machines together implement RPC semantics (i.e. the RPC topology, call semantics and failure semantics). The full call path for a typical RPC is provided in Figure 12. These routines must follow predefined interfaces, and must be built on top of the communication services provided by the runtime library. The implementation of RPC semantics is organized asymmetrically for the client and server programs. On the client side, two routines can be imported, one

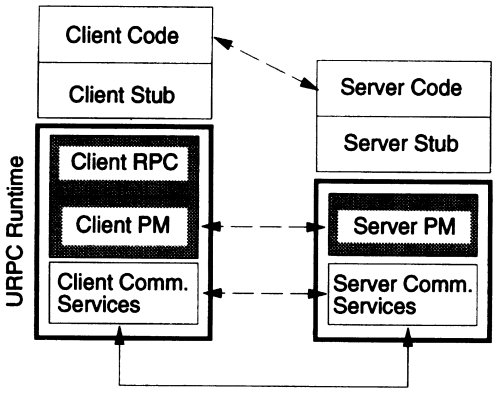


**FIGURE 1.** Architecture of the URPC runtime library.

(*client_pm*) for implementing the client protocol machine, and the other (*client_rpc*) for assembling and disassembling messages to interface with *client_pm*. On the server side, only one routine (*server_pm*) is needed for importing the implementation of the server protocol machine. No separate routine for assembling/disassembling messages is provided for the server because this functionality is factored out and is embedded within the implementation of the server protocol machine, which represents a fan-in point.

An asymmetrical design is preferable since customized features are best handled at different layers on the client and server sides, respectively. Consider the transaction example in Subsection 1.1, in which a batch of requests are buffered until the `FINISH` is issued. The `FINISH` call must be handled differently from other calls by the RPC runtime. The architecture must allow this type of customization information to be presented and isolated in one place. On the client side, the best place for this is below the stub and above the protocol machine layer, so that the programmer has a chance to customize calls before they get into the protocol machine layer. This decomposition is possible because the application-related processing semantics can be separated from the processing semantics for other messages used in the protocol machine.

However, such customization is best handled at a lower layer on the server side, since this sort of separation and decomposition is generally not possible. Server customization to deliver the batch of transactions all together must be handled at the protocol machine layer to allow message buffering before delivery. Such customization semantics cannot be isolated easily from the semantics shared with other messages, so we fold them into the protocol machine layer. Also, the servers generally represent fan-in points for communication. Thus, in a heterogeneous environment, individual clients will be customized to conform to the server protocol, but the server need not be customized separately for interaction with individual clients. Thus, each client will need a separate protocol machine, but the server uses the same machine to interact will all its clients.

The URPC architecture can be symmetrical if it is used for peer-to-peer communication. Each side will have both the client and the server pieces, resulting in a symmetrical architecture.

The communication services component comprises routines to facilitate point-to-point communication between RPC participants (clients and servers). Communication between two end points is modelled using a generic send/receive model.

There are several reasons for this choice of model. First, it is simple and flexible, and any RPC protocol machine can be modelled by a sequence of message exchanges. Our send/receive model uses sequences of four activities to model message exchanges between two end points: preparing a message, sending a message, receiving a message, and processing a message. These

activities are general enough to model any protocol machine behavior, and the communication services module provides functions that might be used in these activities. Second, URPC is designed to deal with heterogeneities, and therefore assumes only the most elementary communication capability for each participant, particularly since complex capabilities like multicast or broadcast are usually limited to LAN environments (excepting for the MBONE). Many sites sometimes even disable these features to reduce unwanted network traffic. Although assuming multicast or broadcast may result in better performance in some cases, our sense is that support for multicast or broadcast should be deferred until there is stronger justification for a more complex design. More complex communications schemes are presently simulated using our basic send/receive communication model.

Routines are provided for initializing, managing, and destroying internal end-point data structures, for binding end-points, for sending and receiving messages, for mapping messages to operations, for converting data to/from the external data representation, and so on.

*Communication handles*

The most important data structure in the runtime is the *communication handle*. Communication handles represent point-to-point communication, and serve as the interface for programmers to customize the runtime. Thus, the handle structure is designed with high configurability in mind. Internally, the handle has the structure shown in Figure 2, consisting of the following parts:

- *Incoming and outgoing message queues.* There are two message queues in each handle for buffering the incoming and outgoing messages. A set of functions is provided for managing these queues and for implementing different message buffering schemes.
- *Communication primitive interface.* The communication-primitive interface provides a transport-layer-independent interface to programmers. It is defined through the set of functions open( ), send( ), recv( ) and close( ), that must be provided when a new transport protocol is introduced. Open( ) does initialization and performs end-point association, send( ) sends out a message from the sending-message queue, recv( ) receives a message and puts it in the received-message queue, and close( ) terminates the end-point association and cleans up the handle.
- *Binding information.* The binding information relates to end-point association, including network type, end-point addresses, transport protocol used, etc.
- *RPC information.* The RPC information contains the RPC status of a handle, the type of the handle (for client or for server), timeout period, error status, etc.
- *Client and server protocol machines (PM).* Programmers may associate two protocol machine implementations with each handle, one for client and one for server.[1] Although only one of the protocol machines can be executed at any time, programmers are allowed to switch between these protocol machines or replace them at runtime.
- *Client and server PM working area.* For each handle, private working areas are designated for protocol machine implementations, so that state can be saved and manipulated between calls to the protocol machines.
- *Basic marshalling and unmarshalling functions.* These functions define basic data conversion functions for marshalling/unmarshalling data. By supplying different routines, programmers can change the external data representation for marshalling/unmarshalling the data, or bypass the data conversion if necessary.

**Communication Handle**

| |
|---|
| Incoming and Outgoing Message Queues |
| Communication Primitive Interface |
| Binding Information |
| RPC Information |
| Client/Server PM |
| Client/Server PM Working Area |
| Basic Marshalling/Unmarshalling Functions |

**FIGURE 2.** Handle structure.

In addition to providing high configurability, this data structure also provides a flexible basis for implementing heterogeneous or multi-protocol RPC communication. It can support heterogeneous RPC by allowing a handle to be bound dynamically with different protocols. By bundling together several handles, the data structure can support an RPC requiring simultaneous communication with many others using different protocols. Given a set of client processes that interact in a certain fashion, URPC can be used to construct a generic RPC tailored to the semantics of these interactions. This can serve as a common or 'base' RPC for this heterogenous environment. This is possible because URPC runtime library and primitives allow RPC developers finer control in constructing different RPC semantics. URPC agents can then be constructed to map URPC into application-specific RPC protocols, as described in [3]. Possible uses of URPC for cross RPC are as follows:

| *Appl. 1* | *URPC agents* | *Appl. 2* |
|---|---|---|
| urpc | ⟺ agent ⟺ | rpc2 |
| rpc1 | ⇔ agent ⇔ agent ⇔ | rpc2 |

[1] *Client_rpc*( ) is imported automatically during stub generation. Therefore, no additional function pointer is allocated for *client_rpc* ( ).

**TABLE 1.** Control message types

| *Type* | *Meaning* |
|---|---|
| **start** | Start a session |
| **end** | End a session |
| **ping** | Are you there? |
| **ack** | Acknowledge a message |
| **error** | An error has occurred |

*Message types*

Programmers describe a protocol machine by defining their own message types and the state changes upon receiving a message. A message type can be defined by providing a message-type ID and a function which will be invoked upon receiving a message with this type. There are two classes of messages: the `UDef` messages and the `UDefCtrl` messages. The `UDef` messages are used to invoke application-level functions, and their definition messages are usually automatically generated by the stub generator. To perform an RPC, the client agent simply sends a `UDef` message to the server agent, and the specified server function will be invoked. When the function is executed, the results are sent back to the client using the same message type. The `UDefCtrl` messages are used to provide protocol control functions, out-of-band control messages for example. For the programmer's convenience, the communication services also provide five built-in control message types: `start`, `end`, `ping`, `ack` and `error`. These five message types are summarized in Table 1. A session here is defined as the duration between a `start` and an `end` message, and is identified by a session number.

There are both conceptual and performance reasons for our choice of two different messages types. Conceptually, it is clearer to distinguish control and application-level messages. Application-level messages originate from applications, and responses to them also return to applications. Control messages originate from the protocol machine layer in URPC, and are usually transparent to the application (excepting for situations like errors, when the application must be notified). The distinction between control and application-level messages also allows the URPC runtime to perform some optimization. For example, control message headers are statically pre-built, and reused each time. Also, the decoding logic for control messages is faster than for application-level messages since they have only fixed-length headers and carry no other data.

*2.1.2. Stub generator and IDL*

The design of the stub generator is focused on extending OSF's current DCE/RPC interface definition language IDL [11] to match the flexibility provided by the runtime library. IDL is the language used to define the interfaces of remote procedure calls. A file containing IDL definitions is called an RPC interface specification (RIS), from which the RPC communication code (stubs) can be generated. Although our IDL syntax is similar to OSF's DCE/RPC IDL [11], it has several unique extensions, which will be introduced in the order they appear in an RIS file. An RIS file consists of three parts:

1. *RPC global definition.* The RPC global definition is a list of attributes and values which are used to name an application and provide default RPC setup for the application. The global definition allows the URPC runtime to construct a unique name for an application, which URPC name servers use to register or look up the application. In addition, the global definition provides defaults for initializing a communication handle. These defaults can be overwritten at runtime through URPC library calls. The global definitions have been extended to include attributes for programmers to specify the functions implementing the RPC semantics, so that proper stubs can be generated to interface with these functions. Table 2 lists the attributes allowed in our global definitions and their meanings.
2. *Type definition.* The type definition declares the types that are used in defining RPC signature. It uses the same syntax as *typedef* in C to declare types. No extensions are used for this part of the definition.
3. *RPC signature.* The RPC signature defines the interface of exported RPC operations. We allow programmers to indicate the initiator and the RPC ID of each RPC operation. Explicit indication of the initiator for each RPC operation allows a program to be a client for some RPC operations, and a server for some others. This feature is necessary when implementing peer-to-peer style RPCs or callback

**TABLE 2.** Attributes allowed in global definitions

| *Attribute* | *Type* | *Meaning* |
|---|---|---|
| aptitle | Required | Application title |
| client_pm | Required | Function implementing the client PM |
| server_pm | Required | Function implementing the server PM |
| transport | Required | Transport-layer protocol used |
| client_rpc | Optional | Function setting up special client RPC/topology (default = urpc_client_call( )) |
| version | Optional | Version number of the application (default = 1.0) |

RPCs. To indicate the initiator, programmers simply prefix a tag ([cl] or [sv]) to the interface definition of an RPC operation. For example, the following RPC signature indicates that the RPC *add*( ) can be initiated by the client program, while the RPC *sub*( ) can be initiated by the server program.

```
[cl]error_st add([in] int handle; [in] int x;
                 [in] int y; [out] int *z);
[sv] error_st sub([in] int handle; [in] int x;
                  [in] int y; [out] int *z);
```

The ability to assign the RPC ID allows programmers to overwrite the default ID assignments. In URPC, each RPC operation must be assigned an ID, which has one-to-one correspondence to an `UDef` message type. The assignment is usually automatically generated by the stub generator. However, in some cases, it may be necessary for programmers to map several operations onto one ID. This feature is useful when implementing an RPC that requires multiple calls to complete, such as asynchronous RPC. In general, asynchronous RPC is accomplished by splitting an RPC into a send-request operation and a receive-result operation, with the application continuing to execute after sending out a request to the server. Therefore, to implement asynchronous RPC, programmers may wish to map two client operations into one operation (RPC ID) at the server. The following example illustrates this scenario, where the assignment of an RPC ID is accomplished by annotating the interface declaration of an RPC operation with [@RPC_ID]. Separate RIS files are created for the client and the server respectively.

```
/*asynchronous client RPC signature */
[cl, @1] error_st add_snd([in] int handle; [in] int x;
                          [in] int y);
[cl,@1] error_st add_rcv([in] int handle; [out] int*z);
                          \* server RPC signature*\
[cl, @1] error_st add([in] int handle; [in] int x;
                      [in] int y; [out] int *z);
```

In this example, *add_snd*( ) and *add_rcv*( ) are mapped into the same server operation *add*( ). Therefore, when *add_rcv*( ) is called, the protocol machine can use the RPC ID of *add_rcv*( ) to check whether or not the result has been received. Also, the tag [*cl*] in the server RPC signature is to indicate that *add*( ) will be initiated by the client.

These IDL extensions are designed to provide the flexibility required by the toolkit. Extensions in the global definition introduce the notion of importing implementations of RPC semantics. Extensions in RPC signature allow the stub generator to generate code to support peer-to-peer communication and multi-step RPC operations. The support for peer-to-peer communication complements the message-passing model in communication services, while the support for multi-step RPC operations allows programmers to construct any RPC interface and call semantics that may be required by an application.

### *2.1.3. Name server*

Name service is an important component of any RPC system. The URPC toolkit can function with any name service, since it only requires a lookup and registration service. However, name service becomes a trickier issue in the presence of heterogeneity, since a single name space may not be available, and naming standards are still evolving. If an integrated name space is available, we recommend using it. For the case where one is not available, URPC provides a simple default naming mechanism, and an associated lookup and registration scheme.

The URPC name server maintains the mappings between names and end-points, and provides two services to the outside world: a lookup service and a registration service. Given a server name, the lookup service returns its end-point address. The registration service is used to register a name-to-end-point mapping with the name server. Although all RPC name servers provide these two services, they use different naming conventions. For example, HP/Apollo NCA RPC [12] uses UUIDs to name applications, while SUN RPC [13] combines a program number and a version number for the purpose. In contrast to traditional name servers with fixed name spaces, the URPC name server provides additional mechanisms for programmers to construct their own name spaces.

We have considered two possible designs for name-space construction mechanisms. Both designs are centered around name processing functions, which determine how names are interpreted and matched. One approach is to allow programmers to provide their own name processing functions to the name server. The other is to provide a generic boolean-valued name processing function, which can be customized for different RPC systems. If two names match, the name processing function returns *true*, and *false* otherwise. We choose the second approach because it results in a simpler name server and allows us to provide a uniform name processing function for all applications. The first approach complicates the name-service protocol because it must include support for importing/selecting name processing functions. In addition, we will be more likely to introduce naming heterogeneities. Such naming heterogeneity will reduce the availability of applications and make integration more difficult.

For flexible name-space construction, we provide a structure for programmers to describe their naming

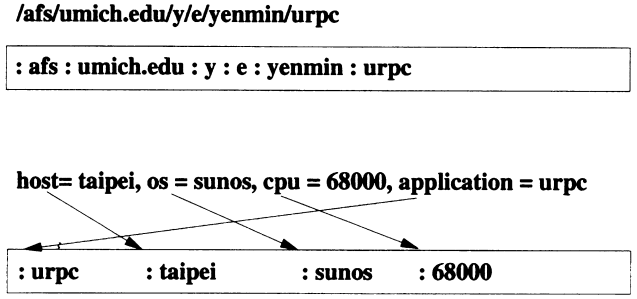


**FIGURE 3.** Mapping hierarchical and attribute-based naming schemes onto URPC name structure.

conventions. In URPC, a name is a unique entity representing an instance of application. A name is represented in a structure called a chain, which is a sequence of bytes partitioned into segments by delimiters ':'. Although this name structure is quite simple, programmers can map both hierarchical and attribute-based naming schemes onto it. With such flexibility, this structure can also be used to support hybrid naming schemes, and can allow multiple naming schemes to coexist. Figure 3 illustrates possible mappings for the hierarchical and attribute naming schemes. To map a hierarchical naming scheme, programmers flatten the hierarchical structure so that each leaf node is uniquely named by its absolute pathname. The absolute pathname is then copied into our naming structure by replacing the '/' with ':'. To map attribute-based naming schemes, programmers simply pre-allocate a segment in the chain structure for each attribute, and the value of the attribute is copied into the segment.

To match two names, the name processing function simply compares them segment by segment. When a wildcard (*) is used in a segment, it will match anything in the corresponding segment of the target name. Wildcards are useful because they allow partial matching on names and reduce the burden of constructing names exactly. For flexibility, the URPC name server allows programmers to specify different comparison operators for each segment, so that numerical relationships (like $<$, $>$, $\leqslant$, $\neq$, etc.) can be checked between two segments. This feature can be useful, for example, when checking compatibility between version numbers.

To support different RPC topologies, the name server supports both single and multiple end-point lookup. For single lookup, the name server returns an end-point matching the given name template (a name plus comparison operators). For multiple lookup, the name server returns all the end-points matched by the name template. Multiple lookup is convenient because it gives application an opportunity to perform its own selection of the end-points, making the binding for broadcast/multicast RPC easier. Programmers can also associate a list of name servers with both types of lookup, so that the lookup will be performed on a group of name servers rather than a single one.

Although programmers can construct a completely new name space, it is often easier to just customize the default name space provided by the URPC runtime. The default name of an application is automatically constructed by the URPC stub generator using the global definition in the RIS file. Each name segment in the default name corresponds to an attribute in the global definition, and programmers can customize a name by appending more segments to the name. The URPC default naming segments are:

*aptitle* : *client_pm* : *server_pm* : *transport* : *client_rpc* : *version* : *user defined segments*

**TABLE 3.** Checklist of URPC extensions

| *RPC parameter* | *Runtime library* | *IDL* | *Name server* |
|---|---|---|---|
| RPC topology | √ | – | √ |
| Call semantics | √ | √ | – |
| Failure semantics | √ | – | – |

## 2.2. Summary of URPC extensions

Table 3 provides a high-level view of URPC extensions designed to provide flexibility in constructing RPC semantics. Extensions in the runtime library must cover all three areas of RPC semantics. For the stub generator/IDL, the extensions need only support call semantics. No extensions are needed for RPC topology because they are embedded in RPC interfaces. Also, no IDL extensions for failure semantics are necessary since they are implicit in RPC call interfaces and independent of the functionalities provided by the stub generator. For the name server, the only extension needed is to support different RPC topologies. No extensions are needed for call or failure semantics, because they are independent of the name service.

Table 4 summarizes the specific features used by the URPC toolkit. The features provided in the runtime library facilitate implementation of RPC semantics (i.e., the protocol machines). The implementation of RPC semantics is based on a generic send/receive model, and so it can express any protocol machine behavior. Since messages may need to carry user-defined information, the runtime library supports interfaces to allow programmers to extend the default message format.

The extensions provided in the stub generator/IDL are designed to support different call semantics. These extensions help in specifying the direction of an RPC and in realizing multi-step RPCs. The direction of an RPC is specified by indicating the initiator of an RPC, thus allowing URPC to support both client/server and peer-to-peer style of RPC communication. Multi-step RPCs can be engineered by mapping RPC IDs between client and server operations.

The extensions to the name server are designed to facilitate lookup service in different RPC topologies. They allow programmers to express how many end-points will be looked up, how end-points should be named, and how

**TABLE 4.** Summary of URPC features

| *Runtime library* | *Stub generator/IDL* | *Name server* |
|---|---|---|
| Import PM implementation for RPC semantics | The indication of initiator to support both client/server and peer-to-peer communication | Single and multiple end-point lookup |
| Generic send/recv model for constructing PM | The RPC ID assignment supports different call semantics | Generic naming structure to accommodate different naming models |
| UDef message types and message format extension | | Customizing name processing function with comparison operators |
| Customizable marshalling and unmarshalling functions | | |

names should be processed. These extensions provide semantics-independent mechanisms for customizing the name services. They correspond to the features of single/multiple end-point lookup, generic naming structure and name processing function customization respectively.

## 3. EXAMPLES

To illustrate the usage of the URPC toolkit, we construct three RPCs: a multicast RPC, a callback RPC and an asynchronous RPC. These were chosen because they are non-traditional RPCs. Each example is presented with its RIS file and client PM implementation. To make the examples easier to understand, client PM implementations are simplified by omitting most error handling code, and the servers export only one or two operations. For the reader's convenience, URPC library calls used in examples are listed in Table 5.

### 3.1. Example 1: Multicast RPC

The multicast RPC in this example is built on top of TCP to multicast messages to servers for display. The messages are delivered to servers with no-return RPC semantics.[2] The RIS file of this example is shown in Figure 4.

The handles representing the end-points of servers are organized into an array, which will be looked up in the client code before a multicast RPC is performed. The array is terminated with a special value INVALID, so that the client PM can check this value to know how many servers to contact. The implementation of the client PM is listed in Figure 5.

[2] The no-return semantics here mean that the server need not reply after receiving client's messages.

**TABLE 5.** URPC library interface used in examples

| *Name* | *Function* |
|---|---|
| urpc_call_serv_op | Executes a function associated with the given RPC ID |
| urpc_error | Sets error status |
| urpc_examine_recv_msg | Examines the message at the head of the recv. queue |
| urpc_get_msg_given_udef | Matches RPC ID with the messages in a queue |
| urpc_get_next_recvmsg | Extracts a message from the head of the recv. queue |
| urpc_get_replyflag | Gets the reply flag in a message |
| urpc_get_rpc_cl_state | Gets the client state associated with an RPC ID |
| urpc_get_udef_type | Gets the UDEF message type (RPC ID) |
| urpc_ioctl | Sets input/output control options |
| urpc_send | Sends a message to the other end-point |
| urpc_set_replyflag | Sets message reply flag |
| urpc_set_rpc_cl_state | Sets the client state associated with an RPC ID |
| urpc_set_send_msg | Puts a message in the send queue |
| urpc_wait | Waits for a period of time before continuing |
| urpc_wait_and_recv | Waits to receive a message |

```
[
aptitle = mdisp;
server_pm = mdisp_sv_pm
client_pm = mdisp_cl_pm
transport = INET_TCP
]
typedef int error_st;
typedef char string_t[1024];

[cl] error_st mdisp([in] int *handleP,
                    [in] string_t ss )
```

**FIGURE 4.** RIS file for the multicast-RPC example.

### 3.2. Example 2: Callback RPC

Callback RPC allows a server to call the client back during servicing an RPC. In this example, the server implements $(x-y)*2$ using the function *sub*( ) ($sub(x,y)=(x-y)$) provided by the client. The RIS file for this example is listed in Figure 6, where different initiator tags are used to indicate that the function *sub_X_2*( ) can be initiated by a client, and the function *sub*( ) can be initiated by a server. Note that initiator tags in IDL do not imply the end-point for the callback. It is up to the server PM to determine which end-point to use to do the *sub*( ) operation. For example, when more than one client may initiate the RPC, the server may choose to direct the callback to a client different from the client initiating the RPC, perhaps because it runs on a faster CPU.

The implementation of the client PM is listed in Figure 7. After sending out messages, the client PM falls into a loop looking for either callback messages or replies from the server (lines 12 to 24). The PM distinguishes messages by checking their udef message types (line 15). If a callback message is detected, the PM calls *urpc_call_serv_op*( ) to execute the requested callback function and sends back the result. If the server replies, the PM returns to the caller and the server's reply will be marshalled.

### 3.3. Example 3: Asynchronous RPC

Asynchronous RPC here is identical to the message-passing style of communication. An asynchronous RPC consists of two operations, one for sending out an RPC request and one for collecting the RPC results. This

```
 1 urpc_msg_t *mdisp_cl_pm(handleP,msg,spare)
 2 int      *handleP;    /* handle array */
 3 urpc_msg_t *msg;      /* outgoing message */
 4 char     *spare;      /* not used */
 5 {
 6     int  err = RPC_OK;
 7     while ((*handleP != INVALID)
                && (err==RPC_OK)) {
 8         urpc_set_send_msg(*handleP,msg);
 9         err=urpc_send(*handleP);
10     }
11     if (err != RPC_OK)
12         return(NULL);  /* error occurred */
13     else
14         return(msg);
15 }
```

**FIGURE 5.** Protocol machine for the multicast-RPC example.

```
[
aptitle = cb;
server_pm = urpc_sv_sr_pm
client_pm = cb_cl_pm
transport = INET_UDP
]
typedef int error_st;

[cl] error_st sub_X_2([in] int handle, [in] int x,
                      [in] int y, [out] int *z);
[sv] error_st sub([in] int handle, [in] int x,
                  [in] int y, [out] *z);
```

**FIGURE 6.** RIS file for the callback-RPC example.

separation allows an application to continue to execute after sending out an RPC request.

To express these call semantics, we must map each exported server operation into a pair of client operations. Although the server stub remains the same for asynchronous RPC, the client stub is different because two operations are needed to complete an RPC instead of one. Therefore, two RIS files are provided to express this situation (see Figure 8).

In Figure 8, a pair of client operations are mapped into one server operation by assigning them the same RPC ID. For example, both *add_snd*( ) and *add_rcv*( ) share the same RPC ID, which maps to the operation *add*( ) in the server.

The client PM must act differently for the send-request and return-result calls (see Figure 9). To distinguish different operations for an RPC, two states (SEND_REQ and WAIT_REPLY) are associated with the send-request and collect-result calls respectively. The state of PM changes when each call is completed (lines 18 and 34).

To handle outstanding asynchronous RPCs, the client PM must match the current RPC ID with the RPC ID in returned messages.[3] When an application requests results for a specific RPC, the client PM first searches the receiving queue to see whether or not the reply message has been received (lines 21 to 22). If the PM has not yet received the reply for the call, the client PM is blocked until the reply message arrives. In the meantime, any other reply messages will be queued in the receiving queue (lines 24 to 32).

The features described in the above examples can also be mixed and matched to provide endless possibilities for supporting multiple RPC semantics in an application. For example, programmers may mix synchronous RPC and asynchronous RPC, so that some of the calls will be asynchronous while other RPCs remain synchronous. Or, programmers may stack these features together to provide more sophisticated RPC semantics, like a multicast asynchronous RPC with callback.

## 4. CICERO: A PROTOCOL CONSTRUCTION LANGUAGE

Cicero is a set of language constructs designed to facilitate the implementation of protocol machines.

[3] To simplify the PM implementation, only one outstanding call for the same RPC ID is allowed.

```
 1 urpc_msg_t *cb_cl_pm(handleP,msg,spare)
 2 int     *handleP;
 3 urpc_msg_t *msg;
 4 char    *spare;
 5 {
 6     urpc_msg_t *reply_msg;
 7     int handle = *handleP;
 8     int ret, rpc_id;
 9
10     urpc_set_send_msg(handle,msg);
11     ret = urpc_send(handle);
12 wait_msg:
13   if (urpc_wait_and_recv(handle) != RPC_OK)
                return(NULL);
14   reply_msg = urpc_get_next_recvmsg(handle);
15   if (urpc_get_udef_type(msg) !=
            urpc_get_udef_type(reply_msg)) {
16      rpc_id = urpc_get_udef_type(reply_msg);
17      /* execute the callback function */
18      urpc_call_serv_op(handle,rpc_id,reply_msg);
19      if (urpc_get_replyflag(reply_msg) == REPLY) {
20         urpc_set_replyflag(reply_msg,NO_REPLY);
21         urpc_set_send_msg(handle,reply_msg);
22         urpc_send(handle);
23      }
24      goto wait_msg;
25    } /* otherwise, it is the result */
26    return(reply_msg);
27 }
```

**FIGURE 7.** Protocol machine for the callback-RPC example.

Our approach to protocol generation is intermediate between full synthesis from very high-level specifications and hand-coding. We start with a constructive specification of the protocol (the protocol *construction*) and translate it to executable code. Cicero is the language in which protocol constructions are written, and provides many features that facilitate this task:

- It provides multi-thread support for protocol execution. Instead of directly providing multi-thread support in the toolkit, we provide multi-thread support through Cicero. This strategy is advantageous because better language support can be provided to facilitate multi-thread protocol implementation.
- It uses *event patterns* [10] to control synchrony, asynchrony and concurrency in protocol execution. Event patterns provide a structured way for programmers to specify relationships between events controlling the protocol execution, so that complex interactions in protocol execution can be organized.

```
[ /* client RIS for asynchornous RPC */
aptitle = asyn_math;
server_pm = urpc_sv_sr_pm
client_pm = asyn_cl_pm
transport = INET_UDP
]
typedef int error_st;

[cl,@1] error_st add_snd([in] int handle,
                         [in] int x, [in] int y);
[cl,@1] error_st add_rcv([in] int handle, [out] *z);
[cl,@2] error_st sub_snd([in] int handle,
                         [in] int x, [in] int y);
[cl,@2] error_st sub_rcv([in] int handle, [out] *z);
```

```
[ /* server RIS */
aptitle = asyn_math;
server_pm = urpc_sv_sr_pm
client_pm = asyn_cl_pm
transport = INET_UDP
]
typedef int error_st;

[cl,@1] error_st add([in] int handle, [in] int x,
                     [in] int y, [out] int *z);
[cl,@2] error_st sub([in] int handle, [in] int x,
                     [in] int y, [out] int *z);
```

**FIGURE 8.** RIS files for the asymchronous-RPC example.

```
 1 urpc_msg_t *asyn_cl_fsm(handleP,msg,spare)
 2 int    *handleP;
 3 urpc_msg_t *msg;
 4 char     *spare;
 5 {
 6   urpc_msg_t *reply_msg = NULL;
 7   int handle = *handleP;
 8   int ret, rpc_id, state, tmp;
 9
10   rpc_id = urpc_get_udef_type(msg);
11   state = urpc_get_rpc_cl_state(handle,rpc_id);
12
13   switch (state) {
14      case SEND_REQ:
15         urpc_set_send_msg(handle,msg);
16         ret = urpc_send(handle);
17         reply_msg = msg;
18         urpc_set_rpc_cl_state(handle,rpc_id,RECV_MSG);
19         break;
20      case WAIT_REPLY:
21         reply_msg = urpc_get_msg_given_udef(handle,
22                        urpc_get_udef_type(msg),RECV_QUEUE);
23         if (reply_msg == NULL) { /* result not recv'd yet */
24 start_recv:
25            if (urpc_wait_and_recv(handle) < 0) break;
26            reply_msg = urpc_examine_recv_msg(handle);
27            if (urpc_get_udef_type(msg) !=
28                        urpc_get_udef_type(reply_msg)) {
29               goto start_recv;
30             } else {
31               reply_msg = urpc_get_next_recvmsg(handle);
32             }
33           }
34           urpc_set_rpc_cl_state(handle,rpc_id,SEND_MSG);
35           break;
36     default:
37         urpc_error(handle,UNKNOWN_CL_STATE);
38         break;
39     }
40     return(reply_msg);
41 }
```

**FIGURE 9.** Protocol machine for the asynchronous-RPC example.

- It helps programmers exploit parallelism in protocol execution. Cicero uses events to provide a dataflow style of execution, which can take advantage of today's multiprocessor architectures by exploiting parallelism in protocol execution. This feature may become increasingly important as more functionalities are pushed into RPC protocols, and as multiprocessor workstations become more common.
- It can hook to existing protocol verification tools. The dataflow style of execution can be translated into other formal models (e.g. Petri nets [14]), making it possible to use existing protocol verification methods/tools.

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 Cicero runtime library can be linked together with the URPC library to add multithread support to the URPC runtime. In this section we will only highlight some features in Cicero, as a complete description of Cicero can be found elsewhere [15].

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 specified events are observed. The `when` construct represents the execution control mechanism as well as the unit of parallelism in Cicero. To perform concurrent execution, programmers simply emit an event instance that can trigger actions in multiple `when` constructs. Table 6 describes semantics of some common event patterns associated with a `when` construct. The `cond` construct implements conditional branches. Our `cond` construct is similar to the LISP `cond` construct (or the switch statement in C), except that when several conditionals evaluate to `true`, the statements associated with all the `true` conditions will be executed in order. The `bundle` is a modularization construct similar to the procedure, and is invoked synchronously. The `bundle` construct defines the extent of visibility for event instances, and provides an environment for sharing variables among a group of `when` constructs. The `escape` construct is used to include C statements in Cicero by enclosing them between '{' and '}'. Therefore, it is the `escape` construct that will encapsulate the calls to the URPC library.

### 4.1. An example

To illustrate the use of Cicero, we will construct an RPC using Cicero and the URPC library. The RPC has at-least-once failure semantics, which means that a

**TABLE 6.** Semantics of various when constructs

| *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$(x , y)$: $A$ end; | Executes action $A$ when either $x$ or $y$ occurs |
| when $(x \wedge y)$: $A$ end; | Executes action $A$ when both $x$ and $y$ occur |
| when $(x \sim y)$ : $A$ end; | Executes action $A$ separately by the sequence of $x$ and $y$ occurrences and in that order |
| 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 |

$|x|$ = number of occurrences of event $x$ to date

service may be executed more than once at the server when failure occurs. The at-least-once failure semantics are specified by the extended FSM (finite state machine) shown in Figure 10, where input events can be associated with conditions.

Although the Cicero specification for the at-least-once semantics can be made more compact, we present a slightly longer version to make the implementation easier to understand. There is a one-to-one mapping of events between the FSM specification in Figure 10 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 shown in the comments within the code segment. All the library functions used in the code segment are also already described in Table 5 in Section 3.

The Cicero code segment is shown in Figure 11. After sending out the message (line 12), two `when` constructs (lines 17 and 21) will run concurrently waiting for a reply or a timeout respectively. If a reply is received, the `bundle` returns. If a timeout 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.

## 5. PERFORMANCE

There are two aspects to the performance gains from URPC. 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

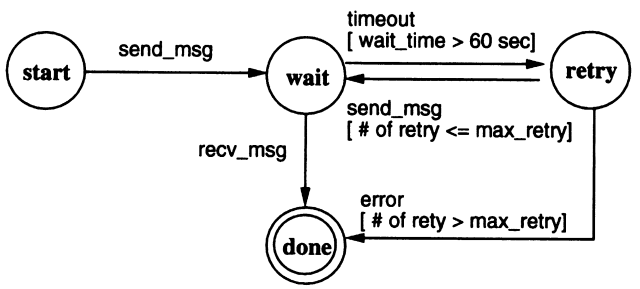


**FIGURE 10.** An extended FSM diagram for at-least-once semantics.

can be significant, as the example in Section 1.1 illustrates.

A second benefit can be a reduction in time for individual calls since customization can eliminate overheads. To facilitate such performance comparison, 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 7 and 8.

We measured the RPC elapsed times for different request sizes and different locations of the server. The size of a request represented the total size of RPC input arguments, and varied from 0 (for null RPC) to 2 kbytes. Two different server locations were used in the measurement: placing the server on the local machine (Local) and placing the server on the remote machine (LAN). All the measurements were carried out on two lightly-loaded SUN Sparc workstations connected through an Ethernet.

The null-RPC elapsed times in the LAN case are smaller numbers than those measured for the local case. This paradox occurs because in the LAN case, some of the work in the client PM is done between the sending and receiving of messages, and in parallel with the server. In contrast, all work in the local case must be interleaved on a single-CPU machine, resulting in longer elapsed time. The speedup in LAN cases disappears as the network delay becomes the dominating factor in RPC elapsed time.

The results show that URPC-ATM1 is about 10% faster than SUN RPC on average, which indicates that an RPC constructed using the URPC toolkit can be just as fast as handcrafted ones. The similarity in performance to SUN RPC is not surprising; it simply reflects the fact that we use a protocol machine implementation similar to SUN RPC.

**TABLE 7.** 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 |

```
 1 bundle ATL1_pm(handle_t handle, urpc_msg_t *msg)
 2 {
 3    int        err;
 4    long       wait_time;
 5    urpc_msg_t *reply_msg;
 6    event      recv_msg, wait, retry;
 7
 8    when (INIT):  /* FSM: start -> wait */
 9     {
10      wait_time = 60;  /* wait for 60 sec. */
11      reply_msg = NULL;
12      urpc_set_send_msg(handle, msg);
13      urpc_ioctl(handle, RECVBLOCK, TRUE); /* block */
14      urpc_send(handle); (* send_msg */
15     }
16     emit recv_msg;
17     emit wait;
18    end;
19    when (recv_msg): /* FSM: wait -> done */
20     { err = (urpc_recv(handle);}
21       cond (err == RPC_OK
22             { reply_msg = urpc_get_next_recvmsg(handle);}
23       end;
24       emit Return:(val=reply_msg);
25    end;
26    when (wait):  /* FSM: wait -> retry */
27      { urpc_wait(wait_time); }
28        emit retry;
29    end;
30    when (retry)*MAX_RETRY: /* FSM: retry->wait/done */
31        { urpc_send(handle); }/* send_msg */
32        emit wait;
33    end: emit Return:(val=NULL) /* RPC failed */
34 }
```

**FIGURE 11.** Cicero code segment for at-least-once semantics.

## 6. RELATED WORK

Much RPC work focuses on designing and implementing new RPC systems to provide new semantics and better performance, as for example in [13, 16–23]. Although our toolkit also provides some popular RPC semantics, it is aimed at providing mechanisms for RPC developers to prototype new RPC systems rapidly. The URPC toolkit achieves its flexibility by letting programmers match the requirements of different RPC semantics by providing their own implementations of RPC semantics and customizing supporting RPC services such as stub generation and name service. None of the existing RPC systems provides such flexibility across its RPC runtime and supporting facilities.

HCS/HRPC [24] is a well-known heterogeneous RPC system. It can support multiple RPC protocols through mixing and matching different implementation of several principle components. However, non-traditional RPC protocols may be hard to emulate with HRPC components cleanly, because its components are designed to model the common functionalities of most RPC facilities. An example given in [24] is an RPC protocol with callback. By giving programmers more control to the runtime, the URPC toolkit not only can facilitate construction of both traditional and non-traditional RPCs, but also can result in a more general solution to the RPC heterogeneity problem [25, 3].

TI-RPC is a transport independent RPC, which can operate on top of any available transport-layer protocols. The URPC toolkit is not only independent of transport-layer protocols, but also independent of RPC semantics. This independence from RPC semantics is achieved by using a generic send/receive model to describe behavior of protocol machines and by allowing customization in its supporting facilities. This independence makes the URPC runtime library highly reusable for constructing new RPC semantics.

The x-kernel [26, 27] is known for configuring protocol stacks with object-oriented sub-protocol components to achieve good protocol implementation. The goal of URPC is different from that of the x-kernel. The URPC toolkit is not for configuring existing protocols, but for constructing new RPC semantics/systems. Therefore, it must give programmers a finer level of control in implementing and customizing RPC systems. Unlike the x-kernel, which deals with the entire protocol stack, the URPC toolkit usually deals with protocol construction above the transport layer. When customized transport-layer protocols are needed, the toolkit allows programmers to import customized transport-layer

**TABLE 8.** 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 |

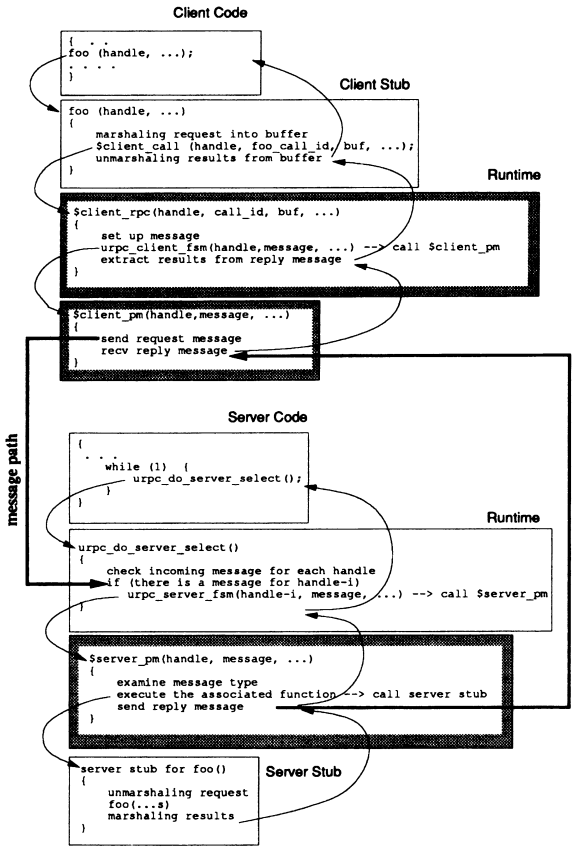


**FIGURE 12.** A typical URPC call path.

protocol implementations through the communication primitive interface in the communication handle.

## 7. CONCLUSIONS

The approach of letting programmers provide the implementation of new RPC semantics appears quite useful in increasing flexibility in constructing new RPC protocols and in reducing coding effort. We enhance flexibility by making it simple for programmers to provide their own protocol machine implementations. New implementations are always required for new RPCs, but the coding effort is reduced with our approach because specifying the protocol machine behavior above the transport layer is the highest level of abstraction possible without restricting innovations in implementation that an RPC might allow. Also, the coding effort for supporting facilities such as stub generators and name servers has been reduced through a customization approach. Therefore, our approach does balance flexibility and coding effort for constructing new RPC systems.

Flexibility in constructing RPC systems can result in efficient RPC implementation. There are two aspects to this efficiency. At a more abstract level, customizing the semantics and behavior of RPC calls can increase the information content of each call, and reduce the number of calls flowing between clients and servers. This is an important performance enhancement. In contrast, systems that do not allow such customization must target the general case, and may require many message exchanges to convey the same information.

In addition, most of the optimization in handcrafted implementations can be preserved by allowing developers to provide protocol machine implementations. Thus, resulting RPCs can be as fast as handcrafted versions. The efficiency of an RPC implementation often depends on several factors: good buffer management, efficient marshalling/unmarshalling, reducing context switches, efficient transport protocols, etc. Although efficient default implementations are provided, the URPC toolkit allows programmers to provide their own implementations for setting up messages, buffering messages, sending/receiving messages, converting data representations, and even new transport-layer protocols. With such flexibility, programmers are able to to apply a variety of optimization techniques to produce efficient RPC implementations. This conclusion is also supported by our experiments and performance measurements in Section 5.

A good protocol construction language is essential to achieve good protocol implementation. Providing correct protocol implementations can be difficult, especially for complex protocols. Cicero is designed to alleviate this problem. Cicero facilitates protocol implementation by providing a better abstraction (event patterns) to control synchrony, asynchrony and concurrency in protocol execution. Also, the dataflow execution model used by Cicero exploits parallelism in protocol execution, and also allows the protocol construction be translated to other formal models to take advantage of existing protocol verification tools. A fuller discussion of these issues in the context of Cicero can be found in [9, 15].

## 8. ACKNOWLEDGEMENTS

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] Tay, B. H. and Ananda, A. L. (1990) A Survey of Remote Procedure Calls. *Oper. Syst. Rev.*, **24**, 68–78.

[2] Ananda, A. L., Tay, B. H. and Koh, E. K. (1992) A Survey of Asynchronous Remote Procedure Calls. *Oper. Syst. Rev.*, **26**, 92–109.

[3] Huang, Y. and Ravishankar, C. V. (1994) Designing An Agent Synthesis System for Cross RPC Communication. *IEEE Trans. Soft. Engng*, **20**, 188–198.

[4] Felten, E. (1992) The Case for Application-Specific Communication Protocols. In *Proceedings of Intel Supercomputer Systems Division Technology Focus Conference*, pp. 171–181.

[5] Maeda C., and Bershad, B. N. (1993) Protocol Service Decomposition for High-Performance Networking. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles*, December. Assoc. for Computing Machinery, Asheville, NC.

[6] 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.

[7] Zayas, E. R. (1991) *AFS-3 Programmer's Reference: Specification for the Rx Remote Procedure Call Facility*. Technical Report FS-00-D164, Transarc Corporation, Pittsburgh, PA.

[8] Chang, R. N. and Ravishankar, C. V. (1984) A Service Acquistion Mechanism for Server-Based Heterogeneous Distributed Systems. *IEEE Trans. Para. and Distrib. Syst.*, **5**, 154–169.

[9] Huang, Y.-M. and Ravishankar, C. V. (1994) Linguistic Support for Controlling Protocol Execution. In *Proc. of 14th International Conference on Distributed Computing Systems*, June, Poznan, Poland, pp. 581–588.

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

[11] Open Software Foundation (1992) *AES/Distributed Computing: Remote Procedure Call*. Cambridge, MA.

[12] Zahn, L., Dineen, T. H., Leach, P. J., Martin, E. A., Mishkin, N. W., Pato, J. N. and Wyant, G. L. (1990) *Network Computing Architecture*. Prentice-Hall, Englewood Cliffs, NJ.

[13] Sun Microsystems (1988) *Remote Procedure Call Protocol Specification Version 2 (RFC 1057)*. Network Information Center, SRI International.

[14] Kavi, K. M., Buckles, B. P. and Bhat, U. N. (1987) Isomorphism Between Petri nets and Dataflow Graphs. *IEEE Trans. Soft. Engng*, **13**, 1127–1134.

[15] Huang, Y. and Ravishankar, C. V. (1993) *Cicero: A Protocol Construction Language*. Technical Report CSE-TR-171-93. Department of EECS, The University of Michigan, Ann Arbor, MI.

[16] Birrell, A. P. and Nelson, B. J. (1984) Implementing Remote Procedure Call. *ACM Trans. Comput. Syst.*, **2**, 39–59.

[17] Gifford, D. K. and Glasser, N. (1988) Remote Pipes and Procedures for Efficient Distributed Communication. *ACM Trans. Comput. Syst.*, **6**, 258–283.

[18] Liskov, B. and Shrira, L. (1988) Promises: Linguistic Support for Efficient Asynchronous Procedure Calls in Distributed Systems. In *Proc. of the SIGPLAN'88 Conference on Programming Language Design and Implementation*, June, Atlanta, GA, pp. 260–267.

[19] Liskov, B. and Scheifler, R. (1983) Guardians and Actions: Linguistic Support for Robust, Distributed Programs. *ACM Trans. Prog. Lang. Syst.*, **5**, 381–404.

[20] Dineen, T. H., Leach, P. J., Mishkin, N. W., Pato, J. N. and Wyant, G. L. (1987) The Network Computing Architecture and System: An Environment for Developing Distributed Applications. In *Proc. of the Summer USENIX Conference*, Phoenix, AZ, pp. 385–398.

[21] Walker, E. F., Floyd, R. and Neves, P. (1990) Asynchronous Remote Operation Execution in Distributed Systems. In *Proc. of 10th International Conference on Distributed Computing Systems*, May, Paris, France, pp. 253–259.

[22] Ananda, A. L., Tay, B. H. and Koh, E. K. (1991) ASTRA—An Asynchronous Remote Procedural Call Facility. In *Proc. of 11th International Conference on Distributed Computing Systems*, May, pp. 172–179.

[23] Yap, K. S., Jalote, P. and Tripathi, S. (1988) Fault Tolerant Remote Procedure Call. In *Proc. of 8th International Conference on Distributed Computing Systems*, San Jose, CA, June, pp. 48–54.

[24] 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. Soft. Engng*, **13**, 880–894.

[25] Huang, Y. and Ravishankar, C. V. (1993) Accommodating RPC Heterogeneities in Large Heterogeneous Distributed Environments. In *Proc. of the 26th Hawaii International Conference on System Sciences (HICSS-26)*, January.

[26] Peterson, L., Hutchinson, N., O'Malley, S. and Rao, H. (1990) The x-Kernel: A Platform for Accessing Internet Resources. *IEEE Comput.*, **23**, 23–33.

[27] O'Malley, S. W. and Peterson, L. L. (1992) A Dynamic Network Architecture. *ACM Trans. Comput. Syst.*, **10**, 110–143.