PEPt
A Minimal RPC Architecture

Harold Carr

Sun Microsystems, 4140 Network Circle, SCA14,
Santa Clara, CA 95054 U.S.A.
harold.carr@sun.com

Abstract. We present a high-level RPC architecture. Although RPC systems seem quite varied they actually share the same fundamental building blocks. We examine the operation of several technologies (e.g., SOAP/HTTP, RMI-IIOP) to show what they have in common, namely four main blocks that we call PEPt: Presentation, Encoding, Protocol and transport. Presentation encompasses the data types and APIs available to a programmer. Encoding describes the representation of those data types on the wire. Protocol frames the encoded data to denote the intent of the message. Transport moves the encoding + protocol from one location to another. The PEPt architecture allows one to understand, use and implement RPC-systems by providing a simple but comprehensive framework in which to place finer-grained details. It may also serve as the basis for RPC infrastructure reuse between seemingly disparate systems. The PEPt architecture *enables an RPC system to adaptively change encodings, protocols and transports.*

1 Introduction

The specification and implementation of Remote Procedure Call (RPC) [1] systems such as DCE [2], distributed versions of C++ [3][4], COM/DCOM [5], CORBA [6], RMI [7], RMI-IIOP [8], XML-RPC [9] and SOAP [10], seems to traverse the same ground repeatedly. One way to avoid reinventing the wheel is to isolate the basic building blocks. This may seem difficult since, at first glance, it may seem RPC systems have nothing in common. We claim that these systems are variations of a fundamental architecture we call PEPt.

We show the PEPt (Protocol, Encoding, Presentation, transport) architecture as a high-level way to structure our thinking, design and implementation of RPC systems. The PEPt architecture has been used in a commercial CORBA system [11]. PEPt embodies the experience of evolving this system from C++ to Java and responding to its changing requirements over time (e.g., alternate protocols and transports, as well as revisions in stubs and encodings). If you build or use more than one RPC system, then the PEPt architecture will help you organize your approach to RPC by providing a clear structure that: provides clarity as to where a function belongs, makes it easier to evolve the system over time, is comprised of a small number of pieces that is easy

to hold in one's head, and specifies a simple decomposition of RPC systems which universally applies.

2 Overview

How can we become better at designing, specifying, building and maintaining RPC systems? We can do so by defining an architecture that is simple enough to hold in the mind as a whole, while being comprehensive enough to describe and implement diverse RPC systems. We show how the PEPt architecture supports the client-side operation of stubs and the server-side operation of ties. We show that the common structure is symmetric: on the client-side a programmer makes a remote call with arguments of specific types (presentation). The types are converted into a representation agreed upon by both the client and server sides (encoding). The encoding is framed with information that carries the intent of the message (protocol). The raw bits of the encoding + presentation are moved from the client location to the server location (transport). The server side goes through these steps in reverse until it obtains the arguments to call the procedure. The whole process repeats itself to return a result. All RPC systems either implicitly or explicitly carry out these steps. PEPt gives us the ability to structure our thinking about RPC systems in such a way as to allow us to build scalable, reusable, maintainable infrastructure.

3 Related Work

The ADAPTIVE Communication Environment (ACE) [12] represents seminal work on applying patterns to network programming. Whereas ACE is a complex system more specific to C++ and high-performance (and, to a lesser extent, CORBA) PEPt is a higher-level, language-independent view of RPC not tied to a particular type of RPC system. PEPt presents an architecture for RPC with fewer "moving parts" in order to guide the overall structuring of a system. ACE focuses more on the details whereas PEPt focuses on the big picture. They complement each other.

RM-ODP's [13] engineering viewpoint channel model is similar to PEPt but does not define important interactions such as how two binding objects interact and connection multiplexing. Although ACE and RM-ODP provide useful patterns they do not give enough detail regarding the overall interaction between the top-level parts of the system.

The SOAP and WSDL [14] specifications allow one to specify different transports and encodings. PEPt is an architecture in which to implement such specifications.

The Jini extensible remote invocation portion of the Jini Davis project [15] is an API and architecture to enable alternate transports and encodings. It is focused on Java's RMI programming model whereas PEPt is language and programming model independent.

Subcontracts [16] were proposed as a way to define new object communication without modifying the base system. PEPt is similar but limits subcontracts to the protocol block, and further delineates that block's relationship to other subsystems to enable finer-grained control and reuse.

PEPt is an architecture for RPC in the same way that others have created architectures and frameworks at different levels of the network such as SASL [17] for security, BEEP [18] for application protocol and Boecking's research [19] in network protocols.

4 Fundamental Building Blocks

The fundamental building blocks of RPC systems are: *Presentation*, *Encoding*, *Protocol* and *Transport*. This paper refers to these blocks as a group as PEPt. Fig. 1 shows PEPt's core architecture.

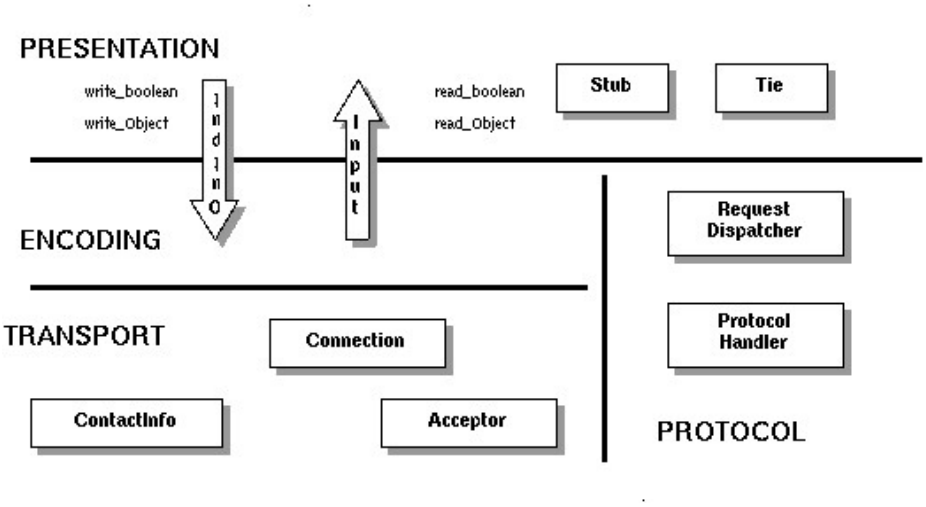

Fig. 1. PEPt architecture with primary interfaces

The boxes in Fig. 1 (e.g., ContactInfo, Input) represent interfaces that represent or bridge the blocks of the core architecture. (Note, the blocks are purposely drawn to *not* suggest a layered architecture.) Each PEPt block is responsible for a key RPC operation.

The **presentation** block includes the APIs used to interact with an RPC system (e.g., stubs/ties), the data types that may be transferred, and error reporting.

We use the term "**encoding** block" to denote the "wire" representation of presentation data types and the conversion process from language representation to wire representation.

Data by itself makes no sense. One needs to include other information along with the data to indicate the intent of the data. The **protocol** block is responsible for

"framing" the encoded data to indicate the intent of the message. On the sending side, the protocol block frames the encoded data with the intent of the message. On the receiving side it interprets the intent. The protocol block says and interprets what a message means.

The **transport** block moves a request or response (i.e., the encoded data and protocol framing) from one location to another. The most common transport today is TCP/IP. CORBA IIOP requests and responses use TCP/IP as their transport. SOAP often uses HTTP as a "transport". However HTTP is a protocol in its own right which uses TCP/IP as its transport. Besides carrying the basic SOAP message (encoding + protocol) HTTP needs its own protocol bits. In general, PEPt views the transport block as a source or sink from which you receive or send bits with no further need for PEPt to deal with additional protocol information. In that case it is clear that CORBA IIOP is a protocol and TCP/IP is a transport. In the SOAP/HTTP case, PEPt would view HTTP as a protocol, framing the SOAP message that, in turn, frames the encoded data. The entire HTTP protocol plus SOAP payload is then given to a TCP/IP transport. PEPt is flexible enough to allow various degrees of coupling between the transport and protocol blocks to handle multiple layers of protocols, as in the SOAP/HTTP case. Once the protocol block is done forming a message it gives it to the transport block to send. Conversely, when the transport block receives a message it gives it to the protocol block for handling. The transport block is responsible for transferring requests and responses from one location to another.

A question naturally arises: why these blocks? Why not more, less or just different blocks? If we look at the related work cited above we can see the subcontract-based architecture in a sense has one block, the subcontract itself. A subcontract is responsible for all variations in protocol, encoding, etc. While a subcontract is a useful pluggability mechanism it does not provide enough structure to help organize the parts that can vary. In other words, subcontracts are too coarse-grained. The ACE architecture goes the other direction: providing multiple "blocks" for network programming. However, ACE's multiplicity is difficult to communicate and easily hold as a whole. We have found, through experience, that PEPt's four main building blocks are a useful division of concerns to answer placement of more detailed functionality and to understand as a whole.

The PEPt architecture is based on our experience with other RPC architectures that tried to completely decouple architectural blocks except for a few well-known interactions. However, when the need to support features such as GIOP fragmentation or SOAP parameters encoded as MIME attachments arose, it was noted that there needs to be a closer coupling between the transport, encoding and protocol blocks. PEPt enables well-known interactions between blocks but also allows private contracts between blocks. For example, if a protocol supports fragmentation, then the encoding block will need to signal the protocol block when the encoding block's internal buffers are full, even though marshaling may not be complete. The protocol block will need to form a fragment message and give it to the transport block to be sent. The PEPt architecture allows such coupling in a generic manner.

Now that we have introduced PEPt's fundamental blocks we continue by showing them processing requests and responses.

5 Request/Response Lifecycle

By following a request all they way through, in detail, on both the client-side and the server-side, it can be shown that PEPt's fundamental blocks provide the right level of granularity to implement RPC systems. We will list the steps necessary to support stub operation and we will show how the PEPt architecture supports those steps.

5.1 Client-Side Lifecycle

The steps to support a remote call are: 1: Get a connection to the service. 2: Get an output stream for the connection. 3: Marshal the arguments into the output stream. 4: Send the arguments to the service. 5: Wait for a response. 6: Get an input stream for the connection. 7: Unmarshal the return value or exception from the input stream. 8: Return normal result or throw exception result. 9: Release any resources used in the remote call. The remainder of this section shows how PEPt supports these steps.

Obtaining a Remote Reference. We do not discuss obtaining remote references in detail here. The main point, in terms of PEPt, is that obtaining a reference generally results in a stub being created in the client. The stub contains the service's address information and code that (un)marshals data from/to the service. (The address information may contain alternate addresses and other information such as transactional and security requirements.) Once we have a stub we can invoke remote procedures (methods).

Invoking a Remote Reference. When a client calls a remote service the client is actually making a call on a stub. A stub is responsible for interfacing with the RPC infrastructure to accomplish the remote call. A stub is part of PEPt's presentation block: the programming model and data types applicable to that model.

Obtaining a Connection to the Service. The stub interacts with the PEPt architecture to service the request. The first step taken is to obtain a connection to the service in order to transport request and replies.

To obtain a connection it is necessary to determine the type of connection and have a factory for the chosen type. To accomplish this the client-side of the PEPt transport block has two main interfaces: ContactInfo and Connection. ContactInfo is an abstract representation of remote references and a factory for Connections. Connection is the interface used to transport requests and replies.

The stub interacts with the protocol block which interacts with ContactInfo to determine information such as location, transport, protocols, encodings, transaction, security, and to create a specific type of Connection. The protocol block interacts with the Connection by sending and getting raw bits transported by the Connection.

(We note that Connection and ContactInfo, along with the Acceptor discussed below, are a form of the Acceptor-Connector design pattern [20].)

Since a Connection may come in many forms: shared memory, Solaris Doors [21], a TCP/IP Connection, ATM, etc., other blocks in the system should not know the specific type of transport being used. In particular, the presentation block should not know anything about the type of Connection. In fact, the type of the Connection (transport), the encoding and the protocol should be able to change dynamically between invocations with no changes necessary at the presentation block. For example, it may be useful to use SOAP/HTTP when an RPC needs to traverse the Internet, but, within an enterprise, using an encoding, protocol and transport that utilizes the internal reliable LAN may be more appropriate.

To obtain a Connection the protocol block interacts with ContactInfo (this protocol block interaction is discussed later). For CORBA this may mean examining an IOR that may contain a TCP/IP host/port pair. Since the CORBA IIOP protocol allows request/reply multiplexing on single connection, an existing Connection may be used or a new Connection may be created if one is not found.

The point at which a Connection is obtained is dependent on the features supported by a specific type of RPC. In RMI-IIOP, Connections are obtained before marshaling because of GIOP fragmentation and Portable Interceptors. (If a GIOP implementation supports fragmentation and if a Portable Interceptor adds service contexts to the GIOP header which overflow the internal buffer containing the encoded header then one or more fragments may be sent. One needs a Connection in order to send a fragment. Thus the Connection must be obtained before marshaling.)

A PEPt implementation of RMI-IIOP would interact with ContactInfo to determine and create the appropriate Connection. In this case, ContactInfo would abstract an IOR. The IOR may contain multiple profiles or tagged components that specify different ways to connect to the service.

PEPt uses the ContactInfo and Connection interfaces of the transport block to enable alternate transports. We will see later how ContactInfo also serves as a factory to enable alternate encodings and protocols. Thus, ContactInfo is the primary client-side pluggability point in the PEPt architecture.

Once we have a Connection to a remote service we need a way to write and read data on the connection. That is discussed next.

Get an Output Object for the Connection. The purpose of a transport block Connection is to carry requests and responses between peers. The actual forming and processing of those requests/responses takes place in other PEPt blocks. To form the request the procedure arguments must be encoded. In other words, there must be a way to convert from the presentation block representation of arguments to the RPC representation (encoding) of those arguments. In PEPt, OutputObject and InputObject are encoding block interfaces that contain and hide the encoding from other blocks. We will discuss how they are obtained and used next.

Once a transport Connection is obtained it is necessary to obtain an OutputObject to be used for marshaling data. One could ask the Connection for an OutputObject, but that would limit the OutputObject to one type of protocol association and it

would limit the Connection to one encoding/protocol combination. Since the remote reference (which is represented in PEPt by ContactInfo) contains the necessary information on what encodings and protocols may be used, it serves as a factory for the OutputObject.

An OutputObject serves several functions. Its interface defines the presentation block data types that may be written to the OutputObject. Its implementation defines the encoding of those types. Its implementation also defines a private contract between the OutputObject and the Connection on how that encoding is stored before being sent (e.g., as an array of bytes).

Once the OutputObject is obtained we can marshal presentation block data into it, which we discuss next.

Marshal the Arguments into the OutputObject. At this level, marshaling is simple. The presentation block stub gives presentation block data types to the encoding block OutputObject to encode and temporarily store in internal storage.

In RMI-IIOP marshaling is actually quite complicated since it must support chunking, fragmentation, indirections, etc. Likewise, SOAP marshaling can become involved in order to support MIME attachments. For example, to support a feature such as GIOP fragmentation PEPt allows encoding block OutputObjects to make private contracts with the protocol block and with the transport block Connection. These contracts enable encoded buffers in the OutputObject to be sent on the Connection before the presentation block is done marshaling.

Marshaling Complete, Send Arguments to Service. After it has finished marshaling arguments, the stub signals the PEPt architecture that request argument marshaling is complete. At this point the encoded arguments (or the last fragment of encoded arguments) need to be sent over the transport. Before the encoded data is actually sent by the PEPt RPC infrastructure it must be framed by protocol information. Protocol framing is the responsibility of the protocol block RequestDispatcher interface. RequestDispatcher is responsible for managing necessary headers (and trailers if present), and for giving the OutputObject's internal encoded data buffers to transport to be sent on the wire.

How do we obtain an appropriate RequestDispatcher? Since ContactInfo abstracts the encoding/protocol/transport combinations available for a specific service it serves as a factory for protocol block objects (as well as transport and encoding block objects).

There is a bootstrap issue here that we will only touch upon lightly. Since the protocol block coordinates interactions between the other blocks, what interface is responsible for initially interacting with ContactInfo in order to choose and create a RequestDispatcher? PEPt handles this by associating a generic RequestDispatcher with the stub. The generic RequestDispatcher's function is to interact with ContactInfo to choose and create a specific RequestDispatcher. Then the specific RequestDispatcher takes over. The specific RequestDispatcher then interacts with ContactInfo to create the Connection and OutputObject.

Generally the choosing and creation of RequestDispatcher, Connection and OutputObject will occur when the stub obtains an OutputObject for marshaling. This is usually the case since protocol information may need to be marshaled into the OutputObject's internal encoded data storage even before beginning argument marshaling. There are two primary examples of the need to create all three block objects at this time. First, if one wants to use one continuous buffer (rather than separate buffers for headers, data, and trailers and the use of scatter/gather IO [22]) the RequestDispatcher needs to write headers into the OutputObject before it is returned to the stub for marshaling. The OutputObject must agree with the Connection on the form of the internal buffer used between them. Secondly, we already mentioned the possibility, in RMI-IIOP, of having interceptors insert service contexts into headers that cause an overflow of the buffer when using GIOP fragmentation. In this case the RequestDispatcher would need to create a fragment message and give it to the Connection for sending even before marshaling begins.

At this point in our discussion we have seen how and when the main interfaces of the four blocks are created and how they are coordinated by the RequestDispatcher protocol block interface to marshal and send a request. We continue by examining how the reply is received and processed.

Wait for a Response. After the request is sent the client-side waits for a response from the server. The operation of waiting for a response is dependent on the protocol in use. PEPt gives the RequestDispatcher control over how to wait for a reply. An HTTP RequestDispatcher will simply block on a read of the Connection on which the request was sent. RMI-IIOP allows message multiplexing on a single Connection. Therefore it is necessary to demultiplex incoming replies. Since different reply messages (and possibly error and close connection messages) can arrive at any time, the RMI-IIOP RequestDispatcher would interact with a ContactInfo factory to create an appropriate protocol block ProtocolHandler object. The ProtocolHandler listens on the Connection for incoming messages (note, issues such as scalability using a "selector" for listening, or "non-listening" transports like Solaris doors are not discussed here). The RMI-IIOP RequestDispatcher would put itself to sleep waiting for the ProtocolHandler to signal that a matching reply has arrived. (Note: the RequestDispatcher and the ProtocolHandler taken together can be viewed as a form of "subcontract" [16].)

Get an Input Object for the Connection. When a reply arrives on the Connection we need to get an InputObject for the Connection so that we can read headers and the remote procedure's result.

When a reply arrives at the Connection it gives the raw bits of the reply to the ProtocolHandler. The ProtocolHandler examines the raw bits to determine the protocol in use (if the Connection is supporting multiple profiles). The ProtocolHandler then asks ContactInfo to create an appropriate InputObject. (Note: a well-designed protocol will use the presentation block data types to read and write headers.)

In the RMI-IIOP case, after the InputObject has been created, the ProtocolHandler reads from it to determine the GIOP version, whether this is the first, continuing or last fragment of a reply or a complete (non-fragmented) reply, and to obtain the request ID. When the reply is non-fragmented or the first fragment of a reply the ProtocolHandler uses the request ID to find the matching request. It then gives the InputObject to the waiting RequestDispatcher and signals it to wake up to handle the reply. When the reply is a continuing or last fragment, the ProtocolHandler uses the request ID to find an existing InputObject (created during the first fragment). It gives the existing InputObject the raw bits of the reply. This forms a producer/consumer relationship between the ProtocolHandler and an existing InputObject.

Once the reply has been matched with a request, the RequestDispatcher will return the InputObject to the stub. The InputObject will be positioned to start reading the marshaled reply (the ProtocolHandler/RequestDispatcher having already read the header information). As noted above, if fragmentation is in effect there will be a private contract between the Connection, the ProtocolHandler and the InputObject such that as more fragments arrive for a particular reply those fragments can be passed to the internal buffers of the InputObject. The InputObject then serves the role of a shared buffer between the stub (consuming the InputObject) and the Connection/ProtocolHandler (filling the InputObject).

Unmarshal the Result and Cleanup Resources. The protocol block RequestDispatcher returns control and an encoding block InputObject to the stub when a reply has been received. The InputObject acts as a bridge between the encoding block and the presentation block.

After unmarshaling, before returning control to user code the stub signals the RPC infrastructure that it may clean up any resources used for this invocation. Example resources are fragment maps that map request IDs to InputObjects, the Input/OutputObjects and Connection used in the request, etc.

5.2 Server-Side Lifecycle

To save space, we will only mention key points regarding the server-side of remote requests. The steps to service a request are: 1: Accept a connection from the client. 2: Receive a request on the connection. 3: Get an input stream for the connection. 4: Find a tie and servant. 5: Use the input stream to unmarshal arguments. 6: Call the servant with the unmarshaled arguments. 7: Get an output stream for the connection. 8: Marshal the result or exception. 9: Send the reply. 10: Release any resources used in the remote call.

Accept a Connection, Receive a Request. The server Acceptor accepts a client's connection request and creates a PEPt transport Connection. When a request arrives the Connection gives the raw bits of the request to its associated Acceptor that acts as a factory for a ProtocolHandler. This gives Connections the ability to handle multiple protocols by delegating the creation of the ProtocolHandler to the Acceptor, which may decode some portion of the initial raw bits to determine the protocol in use and create the appropriate handler.

Get a Request Input Object and Umarshal Header. Once the Acceptor has determined the protocol in use it gives control to the ProtocolHandler that then asks the Acceptor to act as a factory for an InputObject for the Connection.

The ProtocolHandler reads message headers from the InputObject to determine the intent (i.e., type) of the message. The ProtocolHandler may use header information to determine which RequestDispatcher to use to handle the request or it may delegate this determination to the Acceptor . The ProtocolHandler is logically separate from the RequestDispatcher so that if any errors occur during header processing (e.g., header unmarshaling errors) it can form an error reply appropriate for the protocol.

Note that the Acceptor is the server side factory for Connections, ProtocolHandlers, RequestDispatchers, InputObject and OutputObjects. Thus, Acceptor is the primary server-side pluggability point in the PEPt architecture (similar to ContactInfo on the client-side).

Find a Tie, Unmarshal Arguments, Call Servant, Marshal Result. The ProtocolHandler gives control to the RequestDispatcher that finds the appropriate type-specific tie and servant. A tie unmarshals the arguments, calls the servant, then marshals the results.

The presentation block tie gets an OutputObject by interacting with the protocol block RequestDispatcher which, in turn, will interact with the transport block Acceptor and Connection to obtain the correct type of OutputObject. The protocol block may write reply headers into the OutputObject (which may result in fragments of the reply being sent on the Connection). Note that RMI-IIOP's ResponseHandler can be viewed as a standard interface to RequestDispatcher.

Send Reply and Cleanup. When marshaling is complete, the presentation block tie signals the protocol block's RequestDispatcher and ProtocolHandler to resume control. The protocol block takes the encoded framed data and sends it to the client on the transport block Connection: Any resources used while processing the request may then be cleaned up under control of the RequestDispatcher and/or ProtocolHandler.

5.3 Lifecycle Summary

We have shown that the steps taken to invoke and service a remote procedure are essentially the same regardless of the specific presentation block types and APIs, encodings, protocols and transports used. The following tables summarize the blocks and interfaces used at each step. Note that fragmentation may happen any time an output or input object is written or read. We indicate what blocks are involved in fragmentation in rows labeled "(fragmentation)".

Table 1. Client-side steps

Operation	Presentation	Encoding	Protocol	Transport
getConnection			ReqDispatcher	ContactInfo
getOutputObj	stub		ReqDispatcher	ContactInfo
marshalHeader		OutputObject	ReqDispatcher	
(fragmentation)		OutputObject	ReqDispatcher	Connection
marshalArgs	stub	OutputObject		
marshalDone	stub		ReqDispatcher	
sendRequest		OutputObject	ReqDispatcher	Connection
waitForReply			ProtocolHandler	Connection
getInputObject			ProtocolHandler	ContactInfo
unmarshalHDR		InputObject	ReqDispatcher	
(fragmentation)		InputObject	ProtocolHandler	Connection
unmarshalReply	stub	InputObject		
returnResult	stub			
cleanup	stub		ReqDispatcher	

Table 2. Server-side steps

Operation	Presentation	Encoding	Protocol	Transport
getConnection				Acceptor
receiveReqBits			ProtocolHandler	Connection
getInputObject			ProtocolHandler	Acceptor
getReqDispatch			ProtocolHandler	
unmarshalHDR		InputObject	ReqDispatcher	
(fragmentation)		InputObject	ProtocolHandler	Connection
getTie/Servant			ReqDispatcher	
unmarshalArgs	tie	InputObject		
callServant	tie			
getOutputObj	tie		ReqDispatcher	Acceptor
marshalHeader		OutputObject	ReqDispatcher	
(fragmentation)		OutputObject	ReqDispatcher	
marshalResult	tie	OutputObject		
marshalDone	tie		ReqDispatcher	
sendReply		OutputObject	ReqDispatcher	Connection
cleanup			ReqDispatcher	

6 Conclusions and Future Work

RPC specification and implementation could benefit from isolating the key concepts into a core architecture. To this end, we propose PEPt, a four-block design that decomposes RPC systems into presentation, encoding, protocol, and transport blocks. With such an approach, one aspect of the RPC may evolve without disturbing the

others. In other words, when an alternate encoding, protocol or transport is desired there is no need to create another presentation block. Or, alternatively, a new presentation block can reuse existing protocols, encoding and transports.

The PEPt architecture has been used in a commercial CORBA product [11]. That same implementation has been used to prototype a system that supports RMI-IIOP stubs and ties dynamically switching between IIOP and SOAP/HTTP. The core RPC architecture can serve as the basis for understanding, designing, implementing, maintaining and reusing RPC systems.

Although this work makes a fair case for PEPt, space limitations have precluded treatment of important issues such as transactions, security, threads and thread pools, and connection caches. We need to show more detail of each block and other supporting interfaces not shown in this paper, as well as showing finer-grained detail how specific systems are implemented using the PEPt architecture. In particular we need to show how InputObject and OutputObject properly isolate encoding changes from the rest of the system. This is especially important when switching from a self-describing encoding to a binary encoding. The self-describing encoding more gracefully handles extra or missing data. The binary encoding is more compact but may require more agreements between parties. It will be illuminating to measure the throughput and latency for different encodings to answer questions such as: what is the cost of redundancy? What is the cost of self-describing data? We would like to show that invoking SOAP-based web services from a general RPC system need not be difficult. The only real complexity is isolated to where it should be - in the data that is sent back and forth.

We need to show how the protocol block and encoding block deal with the situation where fragments of a request are sent, causing a reply (complete or fragment) to be received even before marshaling is complete. This can happen in RMI-IIOP when the server-side detects an error early in processing or forwards the request to a different object.

Since most underlying protocols are asynchronous we think that PEPt can also serve as the basis for messaging systems, thus unifying our thinking and implementation of those systems. The primary work that needs to be done here is to partition the presentation layer into two dimensions: synchronous versus asynchronous, and message API versus method call. We also need to investigate how the blocks support issues such as messaging store and forward.

7 Acknowledgments

I would like to thank my friend and colleague Ken Cavanaugh at Sun Microsystems for reviewing this paper and, more importantly, being the first person to understand and use the PEPt architecture. Much of the experience and wisdom embodied by PEPt comes from many years of working with Ken. David Ungar, also at Sun Microsystems, asked pointed questions which helped focus the presentation. I had useful interactions with Peter Walker, Hemanth Puttaswamy, Masood Mortazavi, Nick Kassem, Mark Hapner, Roberto Chinnici, Phil Goodwin, Anita Jindal, Arun Gupta

and David Heisser (all at Sun) during the preparation of this paper. Dan Miranker at UT Austin encouraged me to write.

References

1. Nelson, B.J.: Remote procedure call. Ph.D. thesis, Dept. of Computer Science, Carnegie-Mellon University, Pittsburgh, Pa. (1981)
2. The Open Group: DCE. http://www.opengroup.org/pubs/catalog/dz.htm.
3. Carr, H.: Distributed C++. Ph.D. thesis, University of Utah (1994)
4. Kesselman, C., Mani, C.K.: Compositional C++: Compositional Parallel Programming. Caltach (1992) http://caltechcstr.library.caltech.edu/documents/disk0/00/00/01/05/
5. Microsoft: DCOM. http://www.microsoft.com/com/tech/DCOM.asp
6. Object Management Group (OMG): Common Object Request Broker Architecture (CORBA). http://www.omg.org/technology/documents/formal/corba_iiop.htm
7. Sun Microsystems: Remote Method Invocation. http://java.sun.com/products/jdk/rmi/
8. OMG: Java to IDL Mapping. http://www.omg.org/cgi-bin/doc?ptc/02-01-12
9. Winer, D.: XML-RPC. Userland (1999) http://www.xmlrpc.com/
10. Box, D., Ehnebuske, D., Kakivaya, G., Layman, A., Mendelsohn, N., Nielsen, H.F., Thatte, S., Winer, D: Simple Object Access Protocol (SOAP) 1.1. World Wide Web Consortium (2000) http://www.w3.org/TR/SOAP/
11. Sun Microsystems: Corba Technology and the Java 2 Platform, Standard Edition. http://java.sun.com/j2se/1.4.1/docs/guide/corba/index.html
12. Schmidt, D.C.: The ADAPTIVE Communication Environment (ACE). http://www.cs.wustl.edu/~schmidt/ACE.html
13. International Organization for Standardization: Reference Model of Open Distributed Processing. http://www.iso.ch/iso/en/CatalogueDetailPage.CatalogueDetail?CSNUMBER=20696
14. Christensen, E., Curbera, F., Meredith, G., Weerawarana, S.: Web Services Description Language (WSDL) 1.1. World Wide Web Consortium (2001) http://www.w3.org/TR/wsdl
15. jini.org: Jini Davis Project. http://davis.jini.org/index.html
16. Hamilton. G., Mitchell, J.G., Powell, M.L.: Subcontract: A Flexible Base for Distributed Programming. Sun Microsystems (1993) http://research.sun.com/research/techrep/1993/abstract-13.html
17. Myers, J.: Simple Authentication and Security Layer (SASL). Internet Engineering Task Force (1997) http://ietf.org/rfc/rfc2222.txt?number=2222
18. Rose, M.T.: BEEP. http://beepcore.org/
19. Boecking, S.: Object-Oriented Network Protocols. Addision Wesley (2000) http://www.aw.com/catalog/academic/product/1,4096,0201177897,00.html
20. Schmidt, D.C., Stal, M., Rohnert, H., Buschmann, F.*:* Pattern-Oriented Software Architecture, Volume 2. Patterns for Concurrent and Networked Objects. John Wiley and Sons, Ltd (2000) http://siesta.cs.wustl.edu/~schmidt/POSA/
21. Sun Microsystems: Solaris Doors. http://docs.sun.com/db/doc/806-0630/6j9vkb8d1?a=view
22. McKusick, M.K., Bostic, K., Karels, M.J., Quarterman, J.S.*:* The Design and Implementation of the 4.4BSD Operating System. Addision Wesley (1996) 2.6.5 Scatter/Gather I/O. http://www.freebsd.org/doc/en_US.ISO8859-1/books/design-44bsd/x355.html#AEN458