Client-side Encoding, Protocol and Transport Extensibility for Remoting Systems

Harold Carr, **keywords:** middleware, adaptive, reconfigurable, IIOP, SOAP.

***Abstract*—**
Remoting systems (e.g., RPC and Messaging) need to support alternate encodings, protocols and transports, either because of evolving standards or through dynamic negotiation with a peer. Users of a remoting system do not want to be concerned with such details nor have to change programming models just to use a different protocol. They want to concentrate on the data being sent. ContactInfo enables such changes transparently to a programming model and forms the basis for handling fail-over and load-balancing. The key idea is to represent each encoding - protocol - transport combination supported by a peer as a ContactInfo in a list of ContactInfos. When a message is to be sent, one of these ContactInfos is chosen by the remoting infrastructure. The chosen ContactInfo acts as a factory for specific encoders, protocol handlers and connections. If communication fails, another ContactInfo is chosen. We have used ContactInfo to adaptively alternate between XML and binary encoding, protocol and transport combinations. We show size and performance results for these combinations. ContactInfo isolates change from the remoting system user while allowing common remoting infrastructure to be reused.

***Index Terms*—middleware, adaptive, reconfigurable, IIOP, SOAP.**

I. Introduction

REMOTING systems make it easier for programmers to write distributed applications. Some types of remoting systems are RPC [1], messaging, media streaming and group communication. This paper focuses on RPC and messaging (although ContactInfo applies to all types of remoting systems). RPC (or Remote Method Invocation - RMI) systems have a programming model where one invokes a method on an remote object just as one invokes a method on a local object. The details of the communication are handled by the remoting infrastructure. Messaging systems are programmed by adding data to a message structure and then giving that structure to the messaging system infrastructure to send to the receiver. Again, the details are handled by the infrastructure.

There are numerous RPC and messaging systems in existence. For example, Java specifies the RMI, JavaIDL, RMI-IIOP, and JAX-RPC RPC systems and the JMS and JAXM messaging systems. In Java, if one needs to communicate using the WS-I profile [2] one uses the JAX-RPC programming model. If one wants to communicate using IIOP one uses RMI-IIOP. This paper shows that it is unnecessary to have different programming models just to use a particular protocol. This is accomplished by using ContactInfo as a factory for specific protocols.

ContactInfo is a client-side (the role initiating the communication) mechanism that enables a single programming model to be used to communicate over a variety of encodings, protocols and transports (EPT). It may be used to structure new remoting systems or to enable existing systems, like those above, to support evolving standards (e.g., JAX-RPC switching from SOAP-encoding to Doc-Literal) or a non-standard EPT such as JAX-FAST [3].

ContactInfo is the client-side configuration point of the PEPt remoting architecture [4], [5]. The PEPt architecture defines the fundamental building blocks of remoting systems to be: *Presentation*, *Encoding*, *Protocol* and *transport*. This paper refers to these blocks as a group as *PEPt*. Figure 1 shows a block level view of the PEPt architecture. ContactInfo is a factory for specific instances of interfaces in each block.

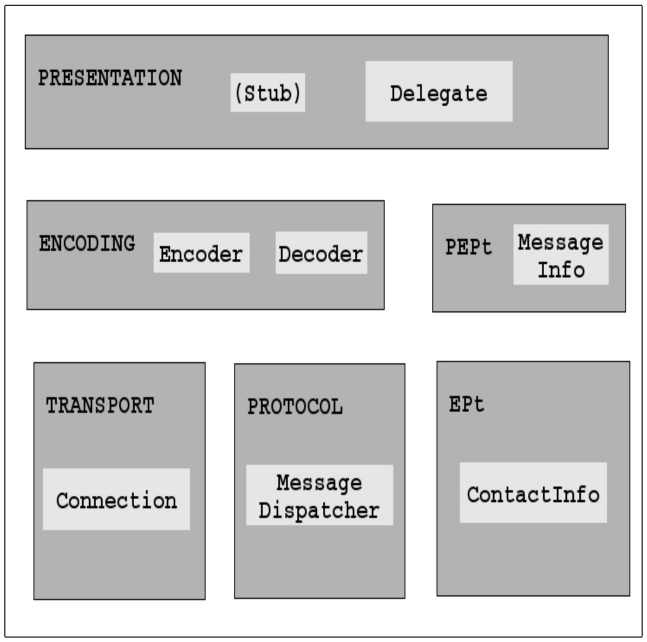

Fig. 1. **PEPt Client-Side Architecture**

The remainder of this paper shows how ContactInfo relates to other middleware systems, shows how ContactInfo fits into the PEPt architecture and shows how ContactInfo operates when handling a message in a remoting system. The focus is on choosing a ContactInfo from a list of ContactInfos and how the chosen ContactInfo acts as a factory for other PEPt interface instances. After showing how ContactInfo works we give performance results of using ContactInfo to support CDR over IIOP, SOAP over HTTP and an ASN.1 binary XML encoding. We finish with conclusions and an outline of future work.

Sun Microsystems, Inc., harold.carr@sun.com

II. Related Work

The ADAPTIVE Communication Environment (ACE) [6] focuses on module details whereas PEPt focuses on the top-level picture in order to guide the overall structuring of a system.

RM-ODP's [7] engineering viewpoint channel model is similar to PEPt but does not define important interactions such as how two binding objects interact.

PEPt is motivated by concerns similar to OpenORB [8]. OpenORB concerns itself with a general component framework and RM-ODP-like bindings within that framework, whereas PEPt considers the framework an implementation detail and gives ContactInfo the binding role of specifying which parts of the communications infrastructure to use on a particular message.

Although ACE, RM-ODP and OpenORB provide useful patterns they do not give enough detail regarding the overall interaction between the top-level parts of the system. PEPt identifies the top-level "blocks" of a remoting system and shows how those blocks interact to adaptively support alternate encodings, protocols and transports.

Subcontracts [9] 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.

The SOAP [10] and WSDL [11] specifications allow one to specify different transports and encodings. PEPt is an architecture in which to implement such specifications.

III. ContactInfo and the PEPt Architecture

PEPt offers a definition of the fundamental blocks of remoting systems. ContactInfo is a factory for specific instances of interfaces in each block. The role of each block in Figure 1 is defined as:

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

The *encoding* block denotes the "wire" representation of presentation data types and the conversion process from language representation to wire representation.

The *protocol* block is responsible for "framing" the encoded data and to indicate the intent of the message.

The *transport* block moves a message (i.e., the encoded data and protocol framing) from one location to another.

The *pept* block is involved in all aspects of the remoting infrastructure while the *ept* block contains ContactInfo that is the factory for encoding, protocol and transport interface instances.

We have found these blocks to be a useful partitioning of remoting system infrastructure that enables a system to support multiple EPTs while giving the ability to reuse common infrastructure (e.g., thread and connection pools).

A. ContactInfo and PEPt Interfaces

A client-side programmer uses a remoting system by way of a Stub (for RPC) or a Delegate and MessageInfo (for Messaging). The Delegate is the access point for the remoting infrastructure and the MessageInfo is the place where data is placed for sending (and receiving). The Stub is an adapter that hides the Delegate and MessageInfo. The Stub transforms an method invocation into a messaging operation by placing the method arguments into the MessageInfo and then using the Delegate to send the message.

ContactInfo represents the address and EPT capabilities of the destination. If the destination has multiple addresses and/or EPT capabilities there will be multiple ContactInfos associated with that destination.

Once a specific ContactInfo is chosen for sending a message, that ContactInfo acts as a factory for messaging infrastructure interfaces. It is a factory for creating an Encoder (and Decoder if a reply is expected) for the specific encoding in the EPT represented by the chosen ContactInfo. Likewise it is a factory for the specific MessageDispatcher that handles the protocol and the specific transport Connnection.

To make this more concrete we continue by showing how ContactInfo operates in the context of sending a message and receiving a reply.

IV. ContactInfo in Action

ContactInfo enables a programmer to concentrate on the data being sent to a remote destination. The details of how that data is encoded and sent should not be an issue. More importantly, the programmer should not need to change programming models just to use a different protocol. That should be handled by the infrastructure.

ContactInfo enables adaptive EPTs and isolates change from the presentation block (i.e., the programmer using the remoting system). ContactInfo is the central extensibility mechanism of the PEPt architecture. We now show how ContactInfo itself is created.

A. Bootstrapping ContactInfo

Suppose a destination is remotely reachable in multiple ways such as CDR over IIOP, SOAP over HTTP and locally reachable via shared memory or Solaris Doors [12]. The destination advertises these capabilities (and the addresses of those capabilities) either by placing that information in an object reference or perhaps by making the information available in a registry. We assume some tool is responsible for reading the reference or registry (or other configuration data) and creating a list of ContactInfos as shown in Figure 2.

The tool, labeled "generator" in Figure 2, creates one ContactInfo for each EPT advertised by the destination. It may also generate a Stub (to adapt to the messaging model) if RPC is being used.

In the rest of this paper we discuss ContactInfo in relation to the messaging model, since we regard RPC as a "layer" on top of messaging.

B. Programmer Interaction

The Delegate contains the APIs used by the programmer (or Stub) to carry out communication. The programmer

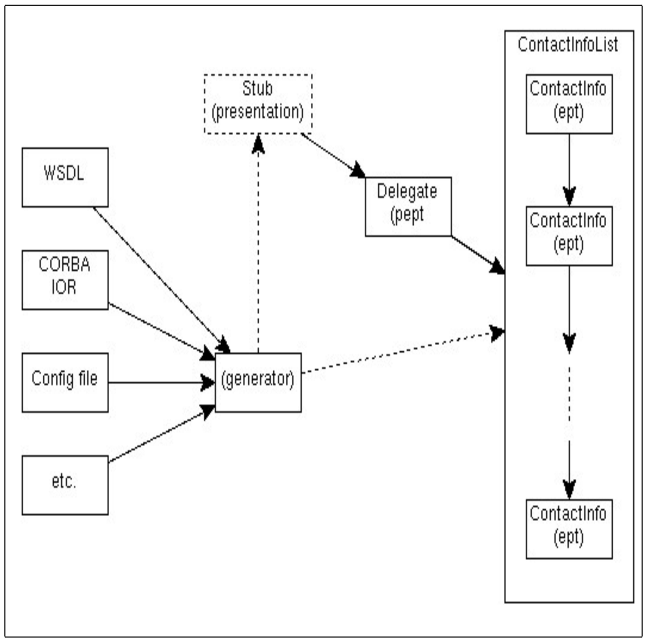

Fig. 2. **Generating ContactInfoList**

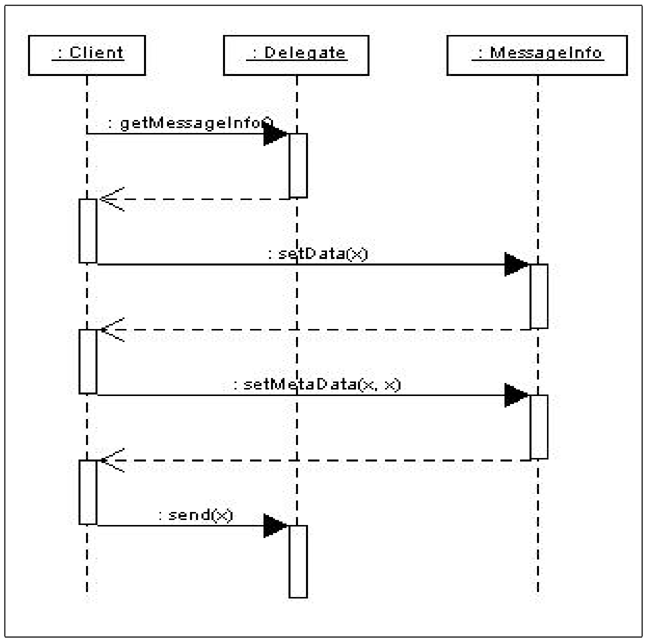

Fig. 3. **Set data and metadata**

(or Stub) interacts with the Delegate to obtain a MessageInfo in which to place data to send to the destination as shown in Figure 3.

Figure 3 shows placing both data and metadata in the MessageInfo. The specific types of data and the kinds of metadata need to be generic since a specific ContactInfo has not yet been chosen. It is possible to choose the ContactInfo before MessageInfo population but we prefer to keep information as generic as possible until as late as possible. The tension between early or late EPT binding is an area of active research.

C. Choosing a ContactInfo

Choosing a ContactInfo is *the* central client-side extensibility point in the PEPt architecture. PEPt provides the ability to plug in an alternate ContactInfo chooser to allow a system to vary its selection policies. The main constraint on a chooser is that it *must* have global knowledge of all possible ContactInfo types (i.e., EPTs). An active area of research is to enable a generic chooser by defining a ContactInfo query or rating system to obviate the need for global knowledge.

A ContactInfo is chosen after the data to be sent has been placed in the MessageInfo and the programmer (or Stub) calls `send` on the Delegate. Figure 4 shows the "choosing" process.

The chooser is encapsulated in an iterator obtained from the ContactInfoList. The (generic) Delegate's function is to call `next` on the iterator to obtain a specific ContactInfo. The details of how `next` is decided are hidden in the iterator. The choosing function can range from choosing the first ContactInfo in the list to more sophisticated load-balancing and quality-of-service mechanisms.

Choosing a ContactInfo is the operation that enables load-balancing (LB), fail-over (FO) and quality-of-service (QoS). A future paper focuses on ContactInfo in relation to LB/FO/QoS so will only briefly mention these other opportunities here.

LB is supported by having the ContactInfo list contain ContactInfos for multiple destinations (e.g., server replicas). A LB policy is then responsible for choosing an appropriate ContactInfo (i.e., destination) for each message. QoS is supported by picking the ContactInfo that guarantees the QoS required by the message. FO is supported by having the remoting infrastructure pick another ContactInfo if communication with a previous pick fails. When handling FO, the Delegate notifies the iterator that communication using the current ContactInfo failed (to help make decisions about future messages) and then the Delegate calls `next` on the iterator to get another ContactInfo.

D. ContactInfo as a Factory

ContactInfo represents an encoding, protocol, transport combination supported by a destination. As such it serves as a factory for a specific Encoder, MessageDispatcher and Connection. This enables a single programming model to be used with multiple EPTs.

D.1 Factory for MessageDispatcher

After the Delegate obtains a specific ContactInfo it uses that ContactInfo to obtain a protocol-specific MessageDispatcher. The Delegate then transfers control to the protocol-specific MessageDispatcher as shown in Figure 5.

D.2 Factory for Connection

Figure 6 shows the responsibilities of MessageDispatcher. The MessageDispatcher uses the chosen ContactInfo to obtain a transport-specific Connection. If the con-

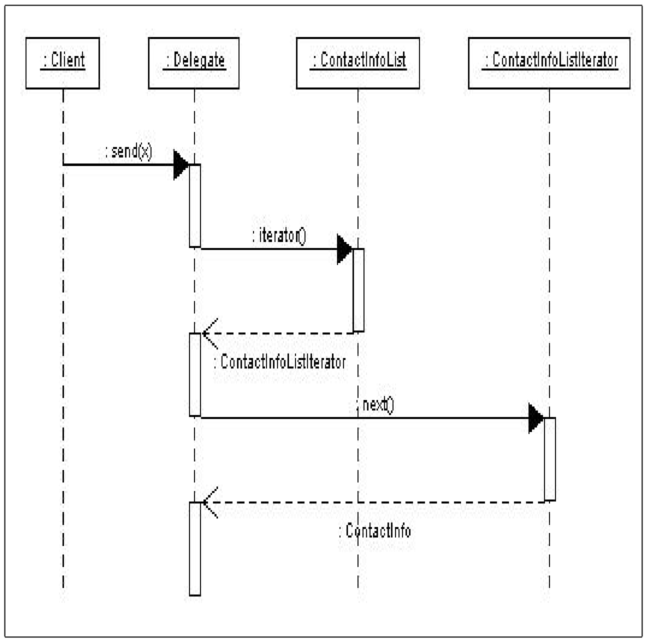

Fig. 4. **Choose ContactInfo**

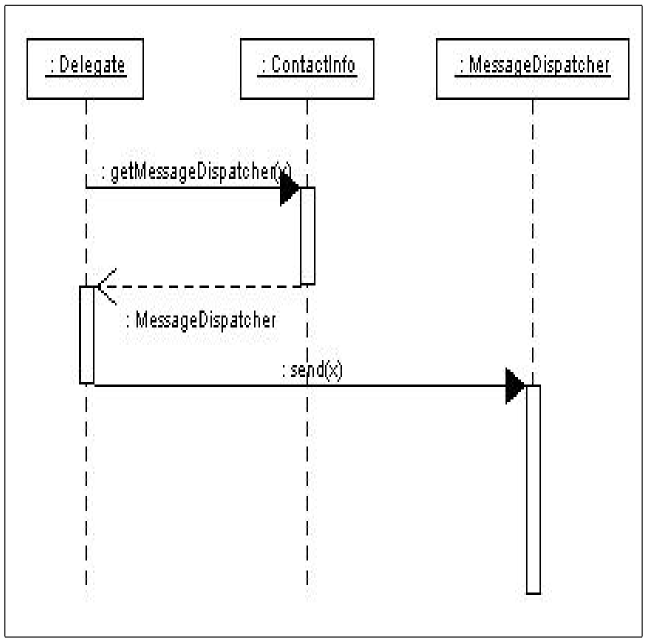

Fig. 5. **Get MessageDispatcher**

nection cannot be obtained an exception returns control to the generic Delegate so it may get another ContactInfo and try again. The exception would only flow out to the client in the case that attempts to obtain a connection for all ContactInfos fails.

Note, the Connection is obtained in the protocol-specific MessageDispatcher rather than the generic Delegate since different protocols support different types of connection management. For example, IIOP allows message multiplexing so if a connection already exists one can be obtained from a cache. Whereas, for HTTP, a new connection is needed for each message. The important point is the ability to select different MessageDispatchers and Connections via ContactInfo rather than specific details of the control enabled by a particular MessageDispatcher or the type of the Connection.

D.3 Factory for Encoder

The MessageDispatcher uses ContactInfo to obtain an Encoder. It gives the Encoder the (meta)data to be encoded and sent. The Encoder can either stream the data as it encodes it (e.g., fragmentation) or it can encode the entire message and then send it on the Connection.

During or after encoding, the encoded data may have other operations applied such as compression, encryption or checksumming. It would be ideal to be able to plug-in encoded-level interceptors independently of Encoders but we have chosen to fold these types of operations into Encoders to avoid traversing the data multiple times. However, this increase in efficiency comes at the cost of reduced code reuse. Once again, the important point is the ability the select different Encoders via ContactInfo independent of the details of other blocks.

E. Handling Replies

If a reply is expected the MessageDispatcher encapsulates the details of waiting for a reply. The details of "waiting" depend on the protocol in use (e.g., depends on the implementation of the specific MessageDispatcher isolated from other blocks). For example, HTTP MessageDispatchers can just block on read then let the stack unwind on reply. Colocated MessageDispatchers can use the client thread to handle the destination dispatch and then unwind on reply. Multiplexed MessageDispatchers (e.g., GIOP) would wait on a condition. A Multiplexed Connection would then use the ContactInfo that created it to obtain the correct multiplexed MessageDispatcher that would read a correlation id to signal the waiting thread. Details such as these are encapsulated in the MessageDispatcher and isolated from the rest of the remoting system blocks to enable the system to adapt to new protocols and to reuse common code.

Once the appropriate MessageDispatcher gains control (via unwind or signal) it uses ContactInfo to obtain a Decoder to decode the reply and put the reply data in the MessageInfo for return to the client. A reply scenario is shown in Figure 7. Again, the important point is to isolate these changing details within the appropriate blocks so the system as a whole may function in a consistent manner despite evolutionary and dynamic changes.

V. Performance

We have used ContactInfo and the PEPt architecture in an RMI system to support CDR/IIOP and SOAP/HTTP. Others have looked at PEPt as a way to integrate ASN.1/HTTP into JAX-RPC [3]. The main point of our work is to show that the blocks of the PEPt architecture represent a useful structuring principle that enables alter-

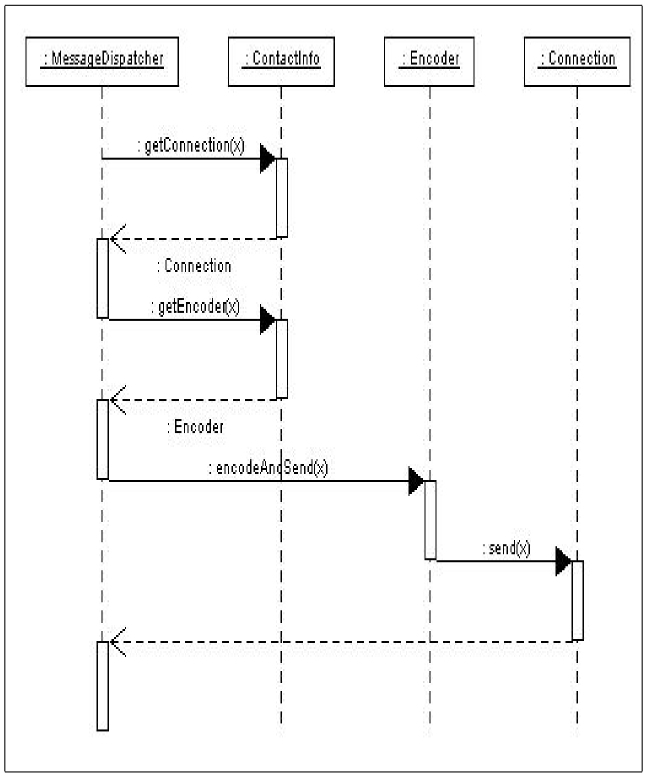

Fig. 6. **Get Connection and Encoder - Send**

nate EPTs such as these to adaptively change in a single consistent system.

Once the ContactInfo factory has created appropriate encodings, protocols and transports the time taken by a remoting system is a function of the work done in each block as shown in Figure 8. Here we see a comparison of a SOAP Doc/Literal encoding using the SOAP protocol over HTTP compared to an ASN.1 Binary XML encoding using SOAP protocol over HTTP. These times were taken in a Java implementation communicating over the local loopback interface. We see the time in transport remains constant whereas the time in protocol and encoding decrease when binary is used. The main point here is that ContactInfo and the PEPt architecture introduce negligible overhead.

Figure 9 shows the time taken to send a message using a variety of EPTs. The important point is the ability to handle alternate EPTs in a single consistent system.

Figure 10 shows the size of the encoded data in the various EPTs. Not surprisingly, the binary representations are smallest with Java's native RMI serialization representation smaller than others such as RMI-IIOP.

The measurements were taken on a dual-processor 400MHz Sparc machine with 512 MB RAM running JDK 1.4.2 when sending 20 instances of the class shown in Figure 11.

VI. Conclusions and Future Work

ContactInfo provides the client-side extensibility point in the PEPt remoting architecture that allows a single

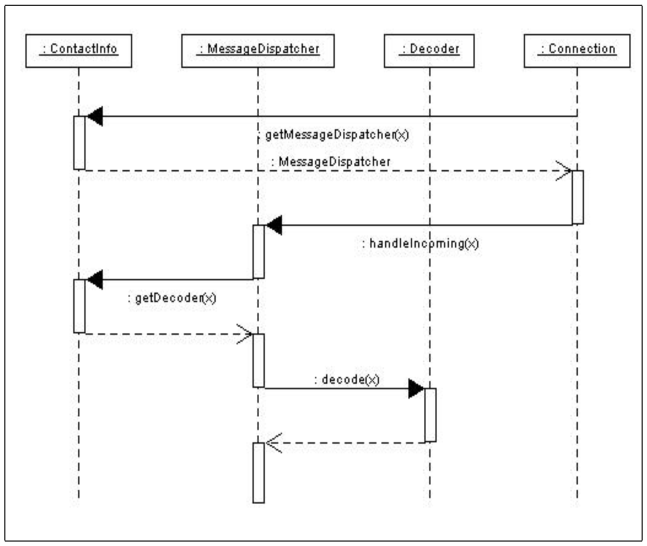

Fig. 7. **Handling Replies**

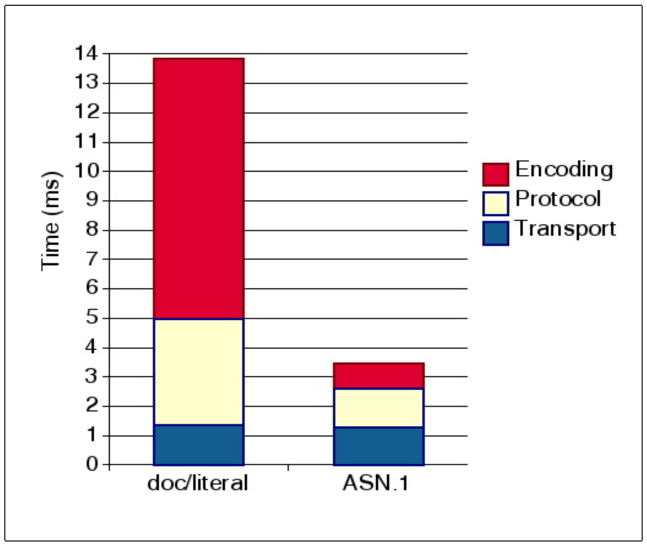

Fig. 8. **Time in PEPt's blocks**

consistent remoting system to evolutionary and adaptively change encodings, protocol and transports. The high-level view of remoting systems given by ContactInfo and PEPt abstracts many details such as component frameworks, threading, thread and connection pools. These are considered (important) implementation details (one can go to ACE and OpenORB for guiding patterns). PEPt instead focuses its attention on the largest building blocks of a remoting system and keeps the number of blocks to a minimum in order to guide the overall structuring of the system.

Future work includes: exploring earlier EPT binding times to support EPT-specific metadata or later times to support increased genericity; creating a generic non-global ContactInfo chooser via an extensible rating system; using PEPt in other remoting systems such as media streaming and group communications. This paper leaves out third-

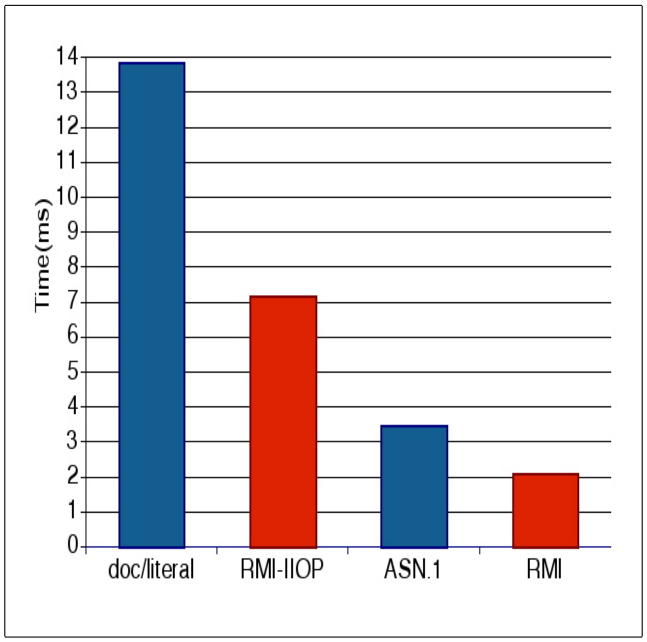

Fig. 9. **Time Comparisons**

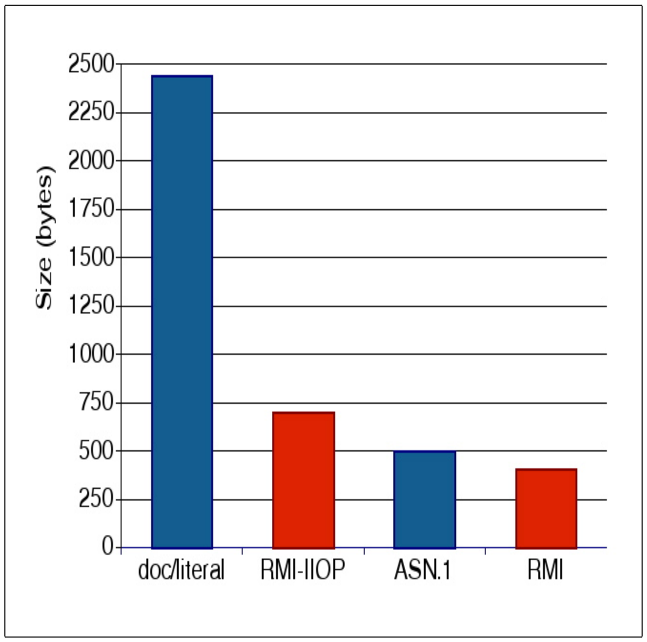

Fig. 10. **Size Comparisons**

party interceptors fit into ContactInfo and the PEPt architecture. We also need to report on PEPt's server-side structuring and extensibility.

We have prototyped systems using ContactInfo for load-balancing, fail-over and quality-of-service. We need to further exercise, measure and develop these systems and report the results. Transactions and security are handled in PEPt via metadata. As noted above, the types of metadata required by specific EPTs compared with the types of data available at the presentation level either restricts the EPTs that may be used for a message or require the presentation level to provide EPT-specific information, which is contrary to the design center of PEPt.

At the present time PEPt uses MessageInfo as the main data passed between the interfaces of the PEPt blocks (i.e., the `x` is the arguments in the UML diagrams). MessageInfo, besides containing the client (meta)data, also contains references to the MessageDispatcher, Encoder and Connection. This makes it easier to add new EPTs that may depend on details of specific interface implementations. For example, to support fragmentation, the Encoder needs to send encoded fragments while it encodes the complete message independent of MessageDispatcher control. However, MessageInfo does not have state, leading to unstructured access to its contents. It would be better to have some control of what is available at particular points in the message dispatch.

We have found PEPt and ContactInfo to be useful for structuring RPC and messaging systems. PEPt is a minimal but comprehensive architecture in which to understand and build adaptive remoting systems.

```
public class Data implements Serializable {
    private String  s;
    private int     i;
    private boolean b;
    public Data(String s, int i, boolean b) {
        this.s = s; this.i = i; this.b = b;
    }
}

new Data("This is a string", 12345678, true);
```

Fig. 11. **Data used in measurements**

VII. Acknowledgments

Santiago Pericas-Geertsen and Paul Sandoz designed the ASN.1 encoding and took the measurements comparing it to other EPTs. Ken Cavanaugh has worked with ContactInfo in the context of CORBA IORs, providing useful feedback in the process. Mark Hapner has encouraged investigating how PEPt can serve in a larger enterprise architecture.

References

[1] Nelson, B. J. *Remote procedure call* Ph.D. thesis, Dept. of Computer Science, Carnegie- Mellon University, Pittsburgh, Pa, May 1981.

[2] Ballinger, K., Ehnebushke, D., Gudgin, M., Nottingham, M., Yendluri, P. *Basic Profile Version 1.0a.* Web Services Interoperability Organization, December 2003

[3] Sandoz, P., Pericas-Geertsen, P., Kawaguchi, K., Hadley, M., Pelegri-Llopart, E *Fast Web Services.* Sun Microsystems, August 2003, http://java.sun.com/ developer/ technicalArticles/WebServices/fastWS/index.html

[4] Carr, H. PEPt - A Minimal RPC Architecture. *OTM Confederated International Workshops HCI-SWWA, IPW, JTRES, WORM, WMS and WRSM 2003 Proceedings*, November 2003, pp 109-122, Catania, Sicily.

[5] Carr, H. One-Page PEPt. *Middleware 2003 Workshop Proceedings*, June 2003, Rio de Janeiro, Brazil.

[6] Schmidt, D. C. Adaptive Communication Environment. http://www.cs.wustl.edu/ schmidt/ACE.html

[7] Reference Model of Open Distributed Processing. International Organization for Standardization, http://www.iso.ch/iso/en/CatalogueDetailPage. CatalogueDetail?CSNUMBER=20696

[8] Parlavantzas, N., Coulson, G., Blair, G. S. An Extensible Binding Framework for Component-Based Middleware. *Proceedings Seventh IEEE Enterprise Distributed Computing*, pp 252-263, September 2003

[9] Hamilton, G., Mitchell, J. G., Powell, M. L. Subcontract: A Flexible Base for Distributed Programming. *Proceedings of the 14th ACM Symposium on Operating Systems Principles*, pp 69-79, December 1993, Asheville, North Carolina

[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

[11] Christensen, E., Curbera, F., Meredith, G., Weerawarana, S. *Web Services Description Language (WSDL) 1.1.* World Wide Web Consortium, March 15, 2001

[12] *Solaris Doors.* Sun Microsystem, Inc., http://docs.sun.com/