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

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

ABSTRACT

Requirement: Users of remoting systems (e.g., RPC and Messaging) want to concentrate on the data being sent. They should not have to use a different programming model just to use a different protocol.

Problem: Remoting systems need to support alternate encodings, protocols and transports, either because of evolving standards or through dynamic negotiation with a peer.

Solution: This paper has two main ideas. First, it clearly partitions a remoting system into four main blocks: presentation, encoding, protocol and transport. It identifies the extensibility points, the responsibilities and interactions of those blocks. Second, it shows how, for each message arriving on a connection accepted and created by an Acceptor, the Acceptor acts as a factory for specific encoders and protocol handlers. Thus a remoting system can dynamically adapt to new encodings, protocols and transports without changing the programming model presented to the programmer.

Experience: We have used this model to adaptively alternate between XML and binary encoding, protocol and transport combinations in an RMI system. We show size and performance results for these combinations.

Conclusion: This model isolates change from the remoting system user while allowing common remoting infrastructure to be extended and reused. The model does not degrade performance.

Categories and Subject Descriptors

C.2.4 [**Computer-Communication Networks**]: Distributed Systems

General Terms

Design

Keywords

Middleware, Adaptive, RPC, RMI, Messaging, IIOP, SOAP

1. INTRODUCTION

One contribution of this paper is to clearly show the major building blocks of a remoting system. The literature often uses phrases like "extensible transports" or "changing protocols." Those phrases, while convenient, are overloaded and too coarse-grained. This paper defines the major blocks of a remoting system to be presentation, encoding, protocol and transport. Once these blocks are defined we use this architecture to enable a uniform programming model (presentation) to be used with remoting system infrastructure that adapts or evolves to changing encodings, protocols and transports.

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. This paper focuses on RPC and messaging systems, although it applies to other types of remoting systems such as media streaming and group communication.

There are numerous RPC and messaging systems in existence. For example, Java specifies RMI, JavaIDL, RMI-IIOP, and JAX-RPC RPC systems and JMS and JAXM messaging systems. In Java, if one needs to communicate using the WS-I profile 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, on the server-side, by using Acceptor as a factory for specific protocols (the second main contribution of this paper).

Acceptor is a server-side (the role reacting to a message) 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 [1].

Acceptor is the server-side configuration point of the PEPt remoting architecture [2, 3], analogous to ContactInfo [4], PEPt's client-side configuration point. 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 server-side architecture. Acceptor is a factory for specific instances of interfaces in each block.

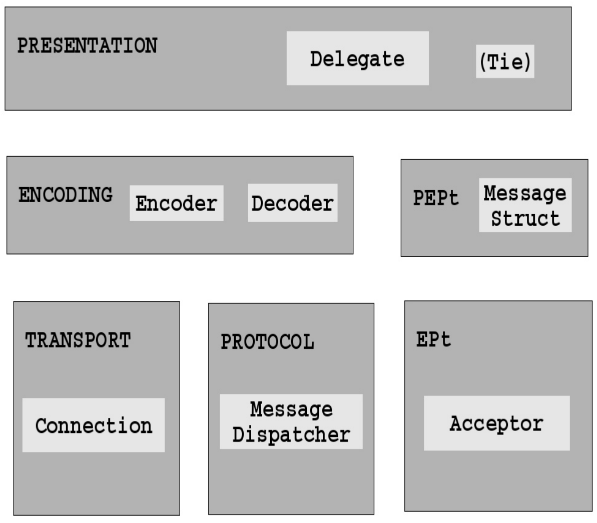


Figure 1: PEPt Server-Side Architecture

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

2. ACCEPTOR AND PEPT ARCHITECTURE

PEPt offers a definition of the fundamental blocks of remoting systems. Acceptor 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 Acceptor 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).

2.1 Acceptor and PEPt Interfaces

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

Acceptor represents the address and EPT capabilities of a server address. If the server has multiple addresses there will be an Acceptor associated with each address. Each address typically handles a single EPT combination, although it is possible to multiplex two or more EPTs at a single address.

When a message arrives at an address an Acceptor associated with the address acts as a factory for messaging infrastructure interfaces. It is a factory for creating a Decoder (and an Encoder if a reply is expected) for the specific encoding in the EPT represented by Acceptor. Likewise it is a factory for the specific MessageDispatcher that handles the protocol, and a factory for the specific transport Connnection.

Acceptor enables programmers to concentrate on their data rather than details of how that data is encoded and sent. Acceptor enables adaptive EPTs and isolates change from the presentation block (i.e., the programmer using the remoting system). Acceptor is the central extensibility mechanism of the PEPt server-side architecture.

3. ACCEPTOR IN ACTION

To make this more concrete we continue by showing how Acceptor operates in the context of receiving a message and sending a reply. (In the rest of this paper we discuss Acceptor in relation to the messaging model, since we regard RPC as a "layer" on top of messaging.)

3.1 Bootstrapping Acceptor

Suppose a server is remotely reachable in multiple ways such as CDR over IIOP, SOAP over HTTP and locally reachable via shared memory or Solaris Doors . For each EPT supported by a server an Acceptor is created that handles incoming connection requests (its traditional role [5]). The server advertises its EPT capabilities (and the addresses of those capabilities) either by placing that information in an object reference or by making the information available in a registry. We assume the server infrastructure allows Acceptors to be registered such that they listen for connection requests from peers as shown in Figure 2. There we see CDR / IIOP / TCP and DocLiteral / SOAP / HTTP EPTs running in a server with a new InfiniBand transport Acceptor being registered to handle CDR or DocLiteral with some undefined protocol.

3.2 Acceptor as a Factory

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

3.2.1 Factory for Connection

When a peer first communicates with the server the Acceptor, acting as a Connection factory, creates a Connection on which messages may flow. This is the traditional role of an Acceptor.

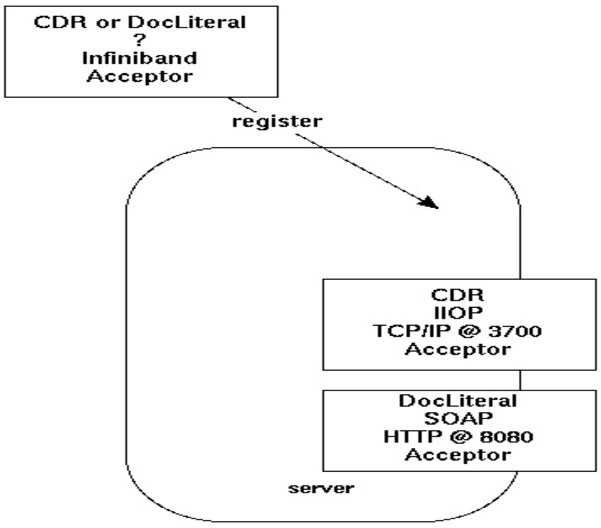


Figure 2: Register Acceptors

When the Connection is created it maintains a reference to the Acceptor that created it such that the Connection may use the Acceptor to act as a factory for MessageDispatchers and Decoders when messages arrive on the Connection.

3.2.2 *Factory for MessageDispatcher*

When the Connection was accepted by the Acceptor it initialized the Connection with a reference to the Delegate that represents the overall PEPt system (it also initializes to contain a reference to its creating Acceptor). When a message arrives on a Connection the Connection first obtains a MessageStruct from the Delegate. The MessageStruct will contain the data and metadata associated with the message, along with references to PEPt interface instances such as the Connection, MessageDispatcher and Decoder used for a message.

The Connection then uses its associated Acceptor to obtain a protocol-specific MessageDispatcher. The Connection then transfers control to the protocol-specific MessageDispatcher as shown in Figure 3.

Note: a multi-protocol Acceptor may need to examine some of the raw bits of the message to determine the EPT in use (or it may delegate that task to a multi-protocol MessageDispatcher). Typically, however, there will only be one EPT per Acceptor.

3.2.3 *PEPt in a Web Container*

Server-side PEPt may be embedded in a web container (e.g., Apache or Tomcat). In this case the underlying web container handles accepting connections and providing data streams to plug-in modules such as servlets. When PEPt is embedded in a web container we assume something like a servlet interacts with the Delegate (representing PEPt) to obtain a MessageStruct in which to place the raw (meta)data of the request (or references to streams) as shown in Figure 4.

The servlet uses metadata from the HTTP headers to determine the appropriate EPtFactory type that is used to create or find an EPtFactory instance. EPtFactory is a base class for Acceptor (and the client-side ContactInfo [4]) that only has methods for creating the EPT interfaces (i.e., minus Acceptor's connection accepting/creation responsibilities).

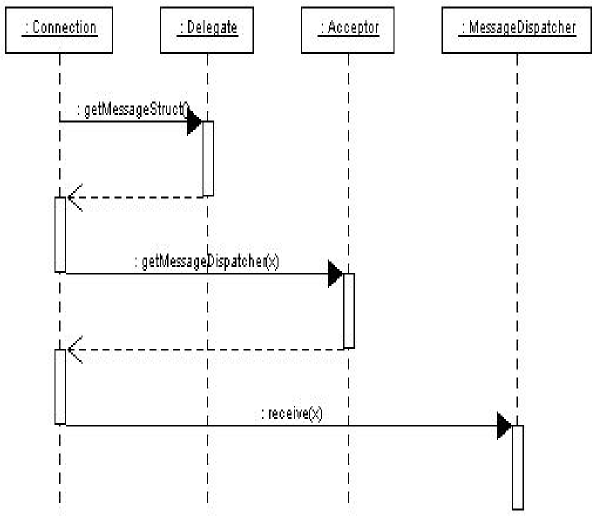


Figure 3: Get MessageDispatcher

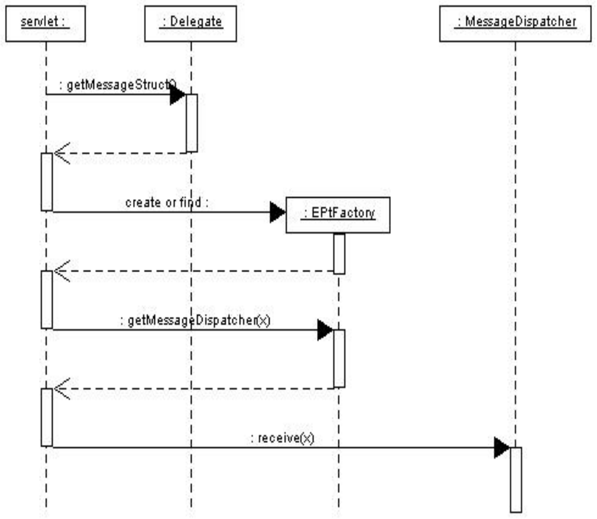


Figure 4: PEPt in a Web Container

The servlet uses the EPtFactory to obtain a protocol-specific MessageDispatcher. The servlet then transfers control to the protocol-specific MessageDispatcher.

It may seem better to get the MessageDispatcher directly, rather by first creating an EPtFactory. However, the remainder of the processing needs a factory to obtain Decoders and Interceptors (and Encoders if there is a reply). By using the EPtFactory the rest of the architecture is isolated from whether the server-side is embedded in a web container or is a stand-alone server managing its own connections.

3.2.4 *Factory for Decoder and Interceptors*

MessageDispatcher uses EPtFactory (or Acceptor) to obtain a Decoder. Decoder decodes the data from the Connection (which may be the web container's data streams). The Decoder either decodes the entire data or makes it available in a streaming, "pull" fashion. If the Decoder decodes the entire data it places the decoded data into MessageStruct.

After decoding, MessageDispatcher uses the EPtFactory to obtain and invoke Interceptors that may reflect on and react to the message. Interceptors are generally protocol-specific such as OMG PortableInterceptors or JAX-RPC handlers. Since the EPtFactory is specific to the protocol it is able to provide a invocation wrapper, Interceptors, around the protocol-specific interceptors. The MessageDispatcher invokes Interceptors with the MessageStruct instance that contains the (meta)data associated with the message as shown in Figure 5.

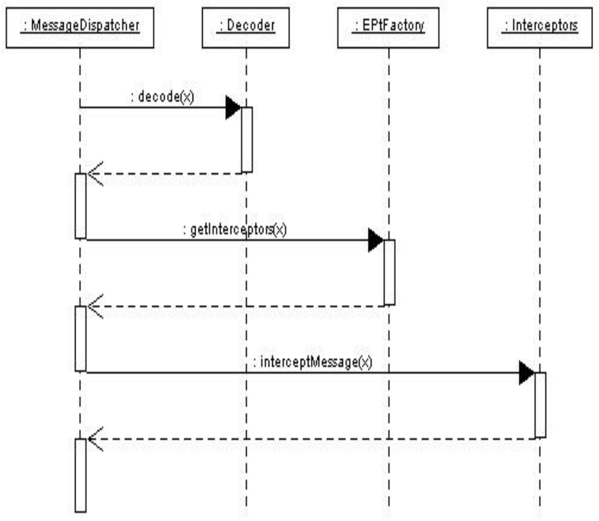


Figure 5: Decode and Intercept

Note: during or after decoding, the decoded 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 Decoders but we have chosen to fold these types of operations into Decoders to avoid traversing the data multiple times. However, this increase in efficiency comes at the cost of reduced code reuse. The important point is the ability the select different Decoders via Acceptor independent of the details of other blocks.

3.3 Finding and Invoking the Target

After interception, MessageDispatcher uses EPtFactory to obtain a protocol-specific TargetFinder. TargetFinder is responsible for finding or creating the appropriate Tie. A TargetFinder may be as sophisticated as OMG's Portable Object Adapter or as simple as JAX-RPC's deployment descriptors. Once the appropriate Tie instance is found the MessageDispatcher transfers control to the Tie as shown in Figure 6.

The Tie gets the decoded data from MessageStruct and calls the "servant" object that contains the actual business logic.

In a messaging system, the operation of finding a target would be to find the appropriate queue on which to place the decoded message (and perhaps to notify a registered waiter of the message arrival).

3.4 Handling Replies

If a reply is expected the MessageDispatcher uses EPtFactory (Acceptor) to obtain appropriate Encoders, Interceptors and Connections with which to send the reply. A server sending a reply is similar to a client sending a request, shown in [4].

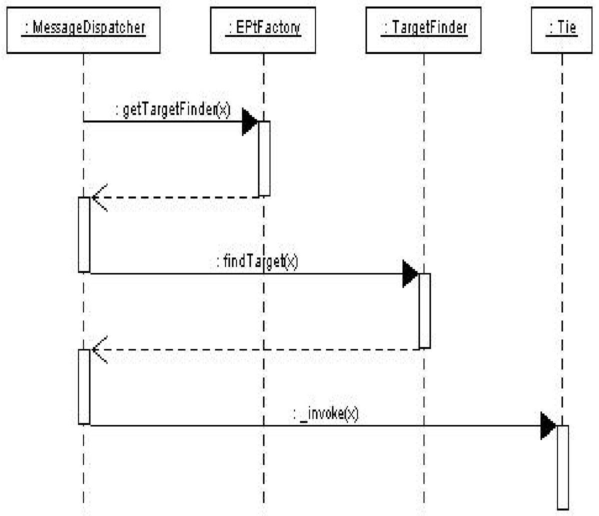


Figure 6: Finding the Tie

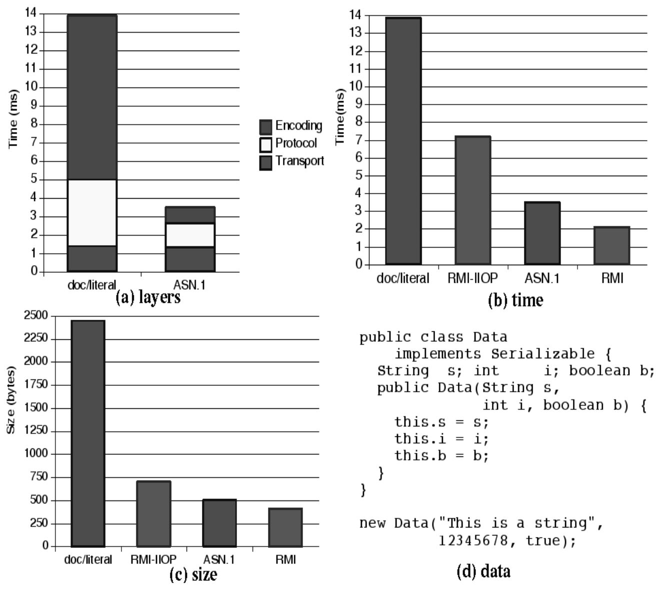
4. PERFORMANCE

Acceptor and the PEPt architecture is used in our CORBA system to support CDR/IIOP and SOAP/HTTP. It is also used to integrate ASN.1/HTTP into JAX-RPC [1]. The main point of our work is to show that the blocks of the PEPt architecture represent a useful structuring principle that enables alternate EPTs such as these to adaptively change in a single consistent system.

Once the Acceptor 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. Figure 7(a) compares Doc-Literal/SOAP/HTTP to ASN.1 Binary XML encoding/SOAP/HTTP. These times (milliseconds) 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 Acceptor and the PEPt architecture introduce negligible overhead. Figure 7(b) 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 7(c) 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 7(d).

5. RELATED WORK

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. OpenORB [8] concerns itself with a general component



```
public class Data
    implements Serializable {
  String  s; int     i; 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);
```

Figure 7: Performance

framework and RM-ODP-like bindings within that framework, whereas PEPt considers the framework an implementation detail and gives Acceptor the binding role of specifying which parts of the communications infrastructure to use on a particular message. 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.

6. OTHER DETAILS

The main observation made by the PEPt architecture is that remoting systems all share in the need to encode data, to frame encoded data with a protocol, and to send the framed, encoded data on a transport. A remoting system built following the PEPt architecture is able to support multiple EPTs without changing programming models.

This observation does not imply that all programming models should be supported, integrated or unified in one programming model. What the PEPt architecture shows is that there is no need to create new programming models just to support different EPTs. It also shows that the same infrastructure may be used to support both messaging and RPC programming models or multiple models of the same type (e.g., RMI-IIOP and JAX-RPC).

PEPt handles sessions, transactions and security via interceptors. The interceptors allow one to add header information such as transaction identifiers or session identifiers to messages. Interceptors that manage session identifiers typically rely on thread-local storage. However, thread-local storage requires implementation guarantees that are not implied by the PEPt architecture.

We have made the claim that PEPt can be used to implement both RPC and messaging systems. However, messaging systems typically involve store-and-forward capabilities that require persistent messages. To implement such a system using PEPt one could create a PEPt MessageDispatcher that would store and retrieve messages from the persistent store. If the persistent store required an encoding different from the transport then one would need a MessageDispatcher that would also act as a internal factory for the persistent encoding since the Acceptor encoding factory is used for transport.

7. PEPT AND SOA

Service-oriented-architectures (SOA) are a step in the right direction towards reusable and adaptive remoting systems in that they focus on the exchange of data. Orchestration and choreography are used to assemble component services and coordinate events (such as message exchanges) between those services. The SOA view tends to be abstracted at the level of data defined via schemas and the "business" processes exchanging, using and transforming the data. SOA does not specify what infrastructure is used to exchange data. It only cares that the exchange takes place.

Although SOA does not specify the remoting infrastructure, the fact is that remoting infrastructure will be used to exchange data. That is where PEPt comes in. A SOA system could use PEPt to implement the lower-level message exchange part of the SOA platform. This gives the SOA platform the ability to integrate with existing services (e.g., CORBA) and to evolve as new encodings, protocols and transports come into use. This integration and evolution can happen without changing the upper orchestration and choreography layers since they are properly using a more abstract view of data and message exchange.

8. CONCLUSIONS AND FUTURE WORK

Acceptor provides the server-side extensibility point in the PEPt remoting architecture. PEPt enables a single consistent remoting system to evolve and adapt to multiple encodings, protocol and transports.

PEPt is used in Sun's implementation of CORBA and JAX-RPC. We have also used PEPt for load-balancing, failover and quality-of-service.

Future work includes using PEPt in other systems such as messaging, media streaming and group communications; managing session information; specifying how to handle multiple encodings in the same message (e.g., encoding for persistence, interceptors and/or remoting); showing PEPt usage in SOA middleware.

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

9. ACKNOWLEDGMENTS

Ken Cavanaugh provided constant critical feedback. Dan Miranker encouraged me to write. David Ungar helped to focus the presentation.

Santiago Pericas-Geertsen, Paul Sandoz, Arun Gupta, Jitendra Kotamraju, Mark Hapner and Anita Jindal have contributed to this work.

10. REFERENCES

[1] P. Sandoz, S. Pericas-Geertsen, K. Kawaguchi, M. Hadley, and E. Pelegri-Llopart. *Fast Web Services.* Sun Microsystems, August 2003,

http://java.sun.com/developer/technicalArticles/WebServices/fastWS/index.html

[2] H. Carr. Pept - a minimal rpc architecture. In *OTM Confederated International Workshops HCI-SWWA, IPW, JTRES, WORM, WMS and WRSM 2003 Proceedings*, pages 109–122, Catania, Sicily, November 2003.

[3] H. Carr. One-page pept. In *Middleware 2003 Workshop Proceedings*, Rio de Janeiro, Brazil, June 2003.

[4] H. Carr. Client-side encoding, protocol and transport extensibility for remoting systems. In *Proceedings of the International Conference on Communications in Computing*, pages 51–57, Las Vegas, June 2004.

[5] D. C. Schmidt. Acceptor-connector. In *European Pattern Language of Programs*, Germany, July 1996.

[6] D. C. Schmidt. 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] N. Parlavantzas, G. Coulson, and G. S. Blair. An extensible binding framework for component-based middleware. In *Proceedings Seventh IEEE Enterprise Distributed Computing*, pages 252–263, September 2003.

[9] G. Hamilton, J. G. Mitchell, and M. L. Powell. Subcontract: A flexible base for distributed programming. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles*, pages 69–79, Asheville, North Carolina, December 1993.