PRACTICAL EXPERIENCES USING THE OMG'S EXTENSIBLE TRANSPORT FRAMEWORK (ETF) UNDER A REAL-TIME CORBA ORB TO IMPLEMENT QOS SENSITIVE CUSTOM TRANSPORTS FOR SDR.

A. Foster
PrismTech
Gateshead UK
Andrew.Foster@prismtech.com

S. Aslam-Mir
PrismTech
San Diego, USA
sam@prismtech.com

ABSTRACT

SDR synthesis using current generation middleware technologies warrants the use of optimum middleware and general software architectures. Key among these is the use of a domain centric yet canonical architecture such as SCA and the use of a open standards in its implementation. The OMG furnishes not only such open standards, but also provides meaningful guidance as to how to efficiently and effectively utilize those standards. One such OMG standard is the Extensible Transport Framework – viz. ETF. Software Defined Radio (SDR) application developers are increasingly exploiting the performance and power potential of different combinations of modern Digital Signal Processor (DSP) devices, FPGAs and general purpose processors (GPP). In an effort to minimize the mips consumption the use of efficient, componentized an re-usable interconnect software abstractions is paramount. Radio hardware implemented with heterogeneous s/w and h/w physical topologies with different permutations, combinations and numbers of such processing elements in them today provide further motivation for the use of such a transport interconnect standard. This paper talks about the practical experiences of implementing such a standards based transport interconnect abstraction in the context of specialized support for synthesizing SDR SCA radios with the aim of maximizing waveform portability.

1. MOTIVATION

In 1998 the members of the OMGs real-time special interest group proposed to the OMG the notion of using custom specialized transport technologies as the transport media underneath a CORBA ORB. This movement took several steps to craft a proposal that would be least controversial to vendors in the OMG who had existing products that did not possess such APIs that were publicly available. The objective of the specification was to establish a framework for plugging in transports in an ORB with sufficient predictability in order to support DRE systems, yet be open enough that anyone could write a custom transport for the ORB much like people could write portable interceptors for the ORBs of the time. The reason to do this was to ensure that the ORBs Generic Inter ORB Protocol (GIOP) could be remapped into some transport technology other than the standard TCP over Ethernet of the time. IIOP (viz. GIOP over TCP/IP) enables reliable remote messaging, however TCP/IP introduces unpredictable latencies unsuitable for many real-time systems, and so the need to plug in lower latency, much more highly predictable transports was seen as a key element in enabling implementations of real-time CORBA ORB applications.

In the context of SDR this issue is even more sensitive owing to the fact that there is currently much debate concerning the use of hardware abstraction layers on platforms in the radio currently perceived to be non-CORBA stations. If these processors or stations are converted to CORBA elements the use of ETF based s/w interconnects over heterogeneous hardware interconnects harmonizes and normalizes the radio platforms architecture internally significantly and enables more plug and plays and faster time to market of any such products.

2. SCOPE OF SPECIFICATION

In scoping out the initial specification an RFP was issued for the specification in which a number of assumptions were made when the RFP was drafted. These included the requirement there was no need to provide support for alternative messaging protocols other than GIOP with CDR encoding. The RFP was however explicit in insisting that there should be a clear separation between the messaging/protocol layers and the transport layers. The intent was that transport plug-in authors could implement ETF interfaces independent of ORB internals, so in addition to the ORB vendors this would facilitate the development of ETF compliant transport plug-ins from third parties including ORB end users, or commercial ETF plug-in suppliers. The following key requirements were also to be addressed:

(1) Provide support for architecture so that a specific transport plug-in could be developed for two different ORBs. Once the plug-in was applied to each ORBs, the application interoperate across the transport successfully.

(2) How exactly the ORB and plug-in interact with each other should be clearly specified and how an ORB actually selects which transport to use should also be specified.

(3) The proposals must support an IOR architecture for non TCP transports such that it is possible for a transport author to create a transport plug-in for two different ORBs that enable application interoperability across the transport via transport specific IOR profiles using clearly identified interfaces and interaction semantics between the ORB and the plug-in.

The work of the submitters came together in 2002 and resulted in the document *Extensible Transport Framework Specification (document reference ptc/04-03-03)* which is an OMG adopted specification. The submitting companies included contributions from Vertel Corporation, Borland Software Corp., Objective Interface Systems, Inc. and Highlander Engineering. Today the specification is presently undergoing the finalization process at the OMG.

3. THE ETF SPECIFICATION

The ETF specification defines a number of mandatory as well as optional interfaces that a plug-in must implement or its author provide. As defined in the specification compliant plug-in implements the following interfaces:

- ☑ `ETF::Profile`
- ☑ `ETF::Connection`
- ☑ `ETF::Listener`
- ☑ `ETF::Factories`
- ☑ `ETF::Handle`

Optional interfaces that the plug-in may choose to implement are those that may be useful if a transport with zero-copy semantics is available and so to conform to the zero copy optional compliance point the interfaces are:

- ☐ `ETF::ConnectionZeroCopy`
- ☐ `ETF::BufferList`

In the next section we shall present the rationale and purpose of each of these interfaces. It is noteworthy that the specification also states that an ETF compliant ORB should implement the `ETF::Handle` interface – making this interface a mandatory compliance point. This compliance point however we believe is somewhat controversial as will be discussed later in this paper.

3.1 ETF::Factories

This is a 'local' interface, which means that in CORBA terms this interface is process local, and cannot have an object reference exported to other objects outside of its local process. It is for ann intensive purposes a local object. This interface provides the 'entry point' for ORB to use the transport. It is plugged into the ORB via a proprietary mechanism identified by IOP::ProfileId (IIOP etc). Indeed the shortcoming of this interface is that fact that although this is used by the ORB to create instances of ETF interfaces, it does not specify how this is plugged into the ORB, this is an implementation detail left to the designer. The methods in the interface are those detailed below which are used to create endpoint listeners, connections, and to demarshall (or extract) profiles characterising specific transports. The methods are:

- ☑ `create_listener(…): ETF::Listener`
- ☑ `create_connection(…): ETF::Connection`
- ☑ `demarshal_profile(…)`

On the server side factories is used to create a Listener objects, whilst on the client side they are used to create connection object and to de-marshal profiles. Usually a factory object is required per protocol, identified by IOP::ProfileId e.g.

- ☐ `TAG_INTERNET_IOP=0` or
- ☐ `TAG_MULTIPLE_COMPONENTS=1`

3.2 ETF::Listener

The Listener interface is again a local interface that handles request for incoming connections from clients and is used to encapsulate the connection establishment protocol. It acts as an ETF connection factory in order to create server side connection objects to which a client will actually connects to. It represents the endpoint which clients contact when connecting an associated `ETF::Profile` endpoint (its transport address). This interface encapsulates connection establishment protocol. Its functionality may be provided by the underlying transport (TCP) or may otherwise be implemented in plug-in code (like for instance in the shared memory transport implementation we present later in this paper). The ORB may use blocking or non-blocking style of call. The ORB thread calls use a blocking `accept()` operation. It is usual for the ETF thread to make calls to the ORB ORB via the `ETF::Handle` callback. In our reference implementation we use the `accept()` method which is called by the ORB and blocks until a connection request is detected, at which point a new Connection object is returned, this is the actual endpoint to which the client connects.

3.3 ETF::Connection

This too is a local interface. It represents a simple transport specific encapsulation of a connection used to read and write byte streams. If the connection semantics are not provided by the transport then they must be implemented in the plug-in, this may include for example ordering, re-tries etc. Client and server process ORBs write and read data to and from transport via a Connection object. The interface also ecapsulates semantics of the actual connection protocol itself also. It may be a reliable, ordered, 1-to-1, bi-directional byte stream. It is an overloaded interface for client and server side. It is usual for the initiation to come from the client-side. The ORB usually creates a connection using factories. It then calls `connect()` to establish connections. The server-side listener creates a new connection object in response to incoming request from client and connects the two connection objects and the endpoints. The client and

server then may read and or write over this new connection.

3.4 ETF::Profile

This is another local interface, which encapsulates the conversion and matching functions used to store transport specific profile data in an IOR. It is is used to hold transport specific address information. This is then used to locate a "matching" profile read from an IOR. The profile holds data related to an address for a transport endpoint. This interface can be used to locate a "matching profile", used on the client side to check whether it is possible to create and use connections supporting a particular protocol/transport or are available, typically when using shared connections. The `marshal()` function in this interface creates an `ETF::AddressProfile` object which packages all profile address data into an octet sequence per profile.

3.5 ETF::Handle

ETF handle is a local interface which is implemented by the ORB. It is only available on the server side and is used by the `Listener` interface to asynchronously inform the ORB about incoming connection requests. It may also be used to asynchronously inform the ORB about the availability of incoming data on existing connections. The sequence of operation is that the ORB registers a Handle with ETF. ETF then makes up-calls to Handle when:

(1) A new connection has been established or
(2) Data has arrived on an existing connection,

The ORB thus avoids some blocking calls to ETF. The ORB must still however make some blocking calls e.g. `connect()` and `write()` still have to be blocking calls

3.6 ETF::ConnectionZeroCopy : Connection

This interface is an optional compliance point provided to support "zero copy" data transfer in the transport where such semantics may be furnished. It requires intelligent buffer allocation to be realised by the underlying transport to be effective, otherwise the extra `BufferList` interface implementation management becomes extra overhead. The interface provides operations to write and read zero copy compatible buffers to and from the transport to the ORB and back – its methods are

- `void write_zc( inout BufferList . . .)`
- `void read_zc( inout BufferList . . .)`

`ETF::BufferList` is a local interface that provides operations that manage the allocation of a memory buffer compatible with the zero copy transport mechanism and used to marshal GIOP protocol into and out of.

3.7 ETF Connection establishment.

Figure 1 illustrates how a client is expected to establish a connection with a server and issue subsequent requests using the specified ETF call sequence.

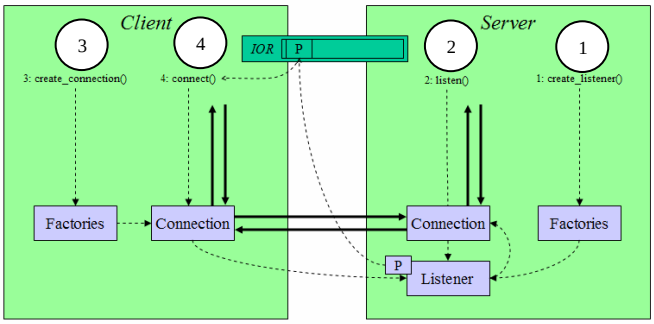


Figure 1:
Illustrating Client – Server interaction where both side ORB utilizes an ETF plug-in. Note the use of the Listener and connections objects and the use of IOR Tagged Profiles to exchange specific transport conduit stacks.

We now describe the sequence used for connection establishment. The ORB issues a create_listener call on the protocol specific factory object passing parameters `RTCORBA::ProtocolProperties`, and `stacksize`, and a `RTCORBA::Priority`. The properties are provided to allow the configuration of protocol specific configurable parameters. Specific protocols have their own protocol configuration interfaces that inherit from the `RTCORBA::ProtocolProperties` interface. If in this process a nil reference for either `ProtocolProperties` is obtained this indicates that the default configuration for that protocol should be used. In general each protocol will have an implementation specific default configuration, that may be overridden by applying the `ServerProtocolPolicy` at ORB scope, for example for TCP there is –

```
local interface TCPProtocolProperties :
                    ProtocolProperties
{
    attribute long send_buffer_size;
    ...
};
```

The ORB then calls `listen()`; this call informs the Listener that the ORB is ready to receive incoming connection requests. At this point the Listener will create a Profile object containing its endpoint address information. An attribute on the Listener gives the ORB access to it. The ORB then either calls

```
set_handle(in Handle .. )
```

which installs a handle with the Listener so that the ORB can receive a call-back when a new connection request is initiated or alternatively the ORB calls `accept()` in which case the call blocks until a client connects to the server. Then a new connection instance is returned. The

ORB now marshals the endpoint information contained in the Listener's profile object along with additional protocol specific information not supplied by the ETF into a complete `TaggedProfile` and publishes the server IOR. On the client side the ORB calls on to the client side factories object in order to perform the converse set of operations, i.e. de-marshal the IOR profile in order to obtain the end point addressing information. The client side ORB then calls the `create_connection` operation on the factory object to instantiate a client side connection object and create the actual underlying socket; once again an `RTCORBA::ProtocolProperties` instance is passed to the operation so as to configure any transport specific properties on the connection. The client side ORB then calls `connect` on the `connection` object, passing the server endpoint address information in a profile object. Usually this `connect` call is also made say on the underlying socket (for the case of TCP) and will be called using the host and port information contained in the Profile object. On the server side the Listener socket listening for the incoming request will detect the connection request and create a new connection object which will actually then be used for the subsequent client server communications. The client side ORB can then call `write` on the connection object passing a CDR encoded byte stream to the connection which in term will be placed on the wire and delivered to the server end of the connection. The ORB on the server side can then issue a `read` on the server side connection object in order to return the marshalled byte stream for the request. The ORB then handles the de-marshalling and local up-call dispatch of the request to target object implementations.

4. EXAMPLE IMPLEMENTATIONS

4.1 TCP Socket Implementation

Figure 2 depicts an example reference implementation of using ETF interfaces implementing a TCP transport plug-in. The purpose is to provide support for Transports other than TCP/IP, but using TCP as an example of how to write a transport plug-in.

The plug-in for a TCP/IP transport is very straight forward and fits into the ETF pattern interfaces nicely. The Listener object simply encapsulates a listening socket, and the the client and server connection objects encapsulate connected sockets. Timeouts on a client connection's `read`, `write` and `connect` operations are implemented by placing the socket into non blocking mode, and then issuing the socket, `read, write` or `connect` calls and then issuing either the standard BSD socket select or poll calls with timeout values set for the calls. The select or poll calls will then wait for the read, write or connect socket event or timeout to complete. It was interesting to note that the ETF specification was silent on the issue of whether or note timeout values are either relative or absolute. Absolute timeouts would be simpler to implement, between ORB and plug-in. Figure 2 illustrates in summary that `ETF::Listener` encapsulates a listening socket, `ETF::Connection` encapsulates a connected socket and `ETF::Profile` encapsulates an endpoint specified by host, and port. This model hides all details of sockets API.

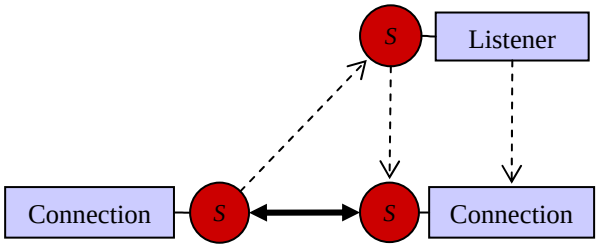


Figure 2:
Diagrammatic depiction of implementing a conventional TCP-IP transport using BSD like socket APIs through the ETF transport framework for a real-time CORBA ORB. (globes with S depict sockets)

4.2 Shared Memory Transport Implementation

The second transport plug-in example we demonstrate is commonly used in SCA and non SCA SDRs today, i.e. the use of shared memory for intra-processor calls. The model ew demstrate is based upon implementing plug-ins for the two shared memory style transports that we've implemented. These transport plug-ins support both a standards based approach using the POSIX APIs and also using System V IPC shared memory APIs. In this example the Listener creates the shared memory segment via a set of `RTCORBA::ProtocolProperties` as before. However, it also creates a Listener control block within the shared memory segment. This listener control block contains information including the client and server send, receive semaphores for the POSIX shared memory transport, or the System V message queue connect and transmit Ids. Other information contained in the block includes the size of the data segment, the size of the data written to the segment and the read offset index and so on. For System V shared memory transport the Profile object addressing scheme is specialized now. It gets converted into a tagged profile and published in an IOR in the form *Shared Memory Segment ID* and host; in this case the host is a string and can either be of the form host IP address or name. For the POSIX shared memory transport object addressing scheme we use the form *Shared Memory Segment Name* and host. In this scenario when a connection request is made by a client the Listener creates a Connection object which in the case of the POSIX shared memory plug-in is allocated in its own connection control block, and has its own set of send and receive semaphores for *request-reply co-ordination* and

handshaking. The System V shared memory transport on the other hand doesn't need to create a connection control block per connection object as each connection object can share and use the message queue setup in the Listener control block. Figure 3 illustrates in summary that `ETF::Listener` creates and manages the shared memory segment, `ETF::Listener` allocates a control block at start, `ETF::Profile` encapsulates an endpoint specified by a file descriptor filename and each `ETF::Connection` get allocated its own control block each. The remainder of shared memory segments is used for transfer buffer, and is shared by the connections. The request response cycle is coordinated and synchronized via either POSIX semaphores, or System V message queues depending on choice of underlying implementation.

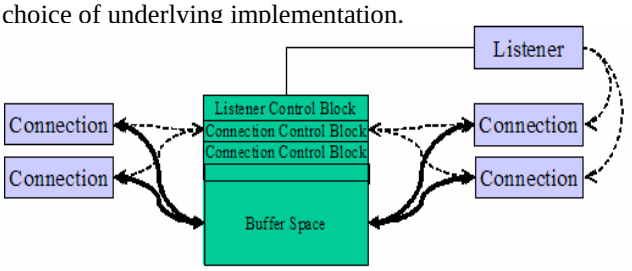


Figure 3:
Diagrammatic depiction of implementing a shared memory transport using shared memory segments again through the ETF transport framework for a real-time CORBA ORB. (Green block depicts shared memory segment)

5. BENEFITS OF SPECIFICATION

As a result of this work it has been possible to produce a single unified transport plug-in API underneath our RTE ORBs (e*ORB SDR C and C++ Editions). In fact it will be possible to use the same plug-ins under our ADA ORBs also. All ORBs share the same pluggable transport layer code implemented once in C. This has meant that we write a transport plug-in once in C and plug into either ORB without excessive porting effort. The has maximised re-use and reduced the amount of test code required. In addition C++ based transport plug-ins developed can be also be appled to the other ORBs, note however that in the case of the C ORB there is no satisfactory approach to handle any exceptions thrown by transport written in C++. The new ETF transport plug-in interfaces are a clear improvement over the original proprietary interfaces that were always vendor specific as there was not standard before. There is a simpler abstraction, which is are easier to document and explain to end users wishing to develop their own.

6. DISCUSSION

Implementing any new specification highlight areas that need judicious interpretation to achieve a solution where the specification falls short. As a result there are always issues with any implementation and interpretation of the specification. We now provide some discussion on some of these.

(1) The separation between the message/protocol and transport layers has not been made clean enough. For example, the Profile interface contains a GIOP version attribute, the transport really shouldn't know anything about message formats as this is inferring GIOP which shouldn't have to be the case. If anything a Protocol version attribute such as `IIOP:Version` would have made more sense.

(2) The Profile's marshal function assumes CDR encoding and doesn't have sufficient information to marshal a complete Tagged Profile.

(3) The specification states "a Factories object needs to have an identifier so that the ORB can select the correct transport type"– but in fact it does not. An `ETF::Factories` object is identified by an `IOP::ProfileId` which in fact is much more than "transport type". An `IOP::ProfileId` identifies a protocol, which implies a specific message layer and tagged profile encoding association.

(4) At present it is not obvious if one can use a transport plug-in with different message layers to form different protocols.

(5) The specification in its current state with the defined interfaces in their unmodified state are problemsatic and need workarounds. In particular the marshalling and un-marshalling functions fall short of the mark since the ETF plug-in has no way of adding tagged components to a tagged profile, and no way of reading them when de-marshalling a tagged profile. This is a key issue logged in the OMGs finalization task force (FTF) is Issue 7594 which states that `ETF::Profile::marshal()` and `ETF::Factories::demarshal_profile()` are not workable. This is owing to the fact that the ETF plug-in has no access to `IOP::TaggedComponents` in the full `IOP::TaggedProfile`. Therefore when marshalling a tagged profile, the ETF plug-in may want to add tagged components; conversely when un-marshalling it may want to read them. When marshaling a full tagged profile,the ETF marshals part of it (`ETF::AddressProfile`), and then the ORB marshals the remainder viz. `IOP::ObjectKey` and `IOP::TaggedComponents`. For IIOP, at least, these two parts must form a single encapsulation. Oddly the responsibility for encoding and decoding a full tagged profile is split between the ORB and ETF. The result is that neither party has access to all the information necessary.

The solution to this problem is to give the responsibility for marshalling and un-marshalling a complete tagged profile to the ETF plug-in, however there is presently no standard representation for a tagged profile, so these interfaces will have to be implemented on a per ORB

basis, as each ORB has its own internal way of representing a tagged profile.

PrismTech alternative approach has been to create an extra ORB specific protocol abstraction which is responsible for marshalling and un-marshalling protocol specific tagged profiles. The ETF in this case becomes only concerned with transport specifics and endpoint addressing then

(6) If the `Factories` interface is extended to support transport identification and versioning, (transport type, version, vendor tag), then it would be possible to have different implementations of the same transport supported for a single protocol. This would enable support for different message formats e.g. *ProfileId = IIOP, transport type = TCP/IP , Vendor Tag = Berkley sockets or BF3Net*

(7) The `ETF::Handle` interface serves two distinct roles:

(a) connection establishment

(b) message arrival

It would have been better to have two separate handle interfaces to encapsulate the different behaviour, this would enable call-backs and blocking for different areas e.g. call-backs for connection establishment and blocking calls for data transfer

(8) The `Handle` interface as specified is always registered with the Listener and is used for two different purposes , either connection establishment or signalling the ORB that data has arrived on a connection. It would have been better if this behaviour had been split between two different types of handles, one of which could could have been installed with a Listener simply for connection establishment, the another handle type could be installed with each specific connection in order to support a reactive model. The current model is one where a Listener with a single handle can support a reactive style of request de-multiplexing.

(9) A handle object can only be installed on the server side. The specification doesn't support the symmetrical installation of a handle on the client side in order to support for example the de-multiplexing of replies from a shared connection or bi-directional GIOP. This behaviour can still be achieved , however the client side ORB must use a dedicated IO thread to do either the blocking or polling calls on the connection. At present since a handle cannot be installed on the client side, this implicitly forces the ORB designer or plug-in implementer wishing to implement Bi-directional GIOP to use private connections. The penalty for this is of course the use of many more threads on the client side.

7. CONCLUSIONS

The ETF specification provides a standard set of interfaces through which a transport other than TCP/IP could be plugged into the ORB. It is felt at present given relevant implementation experience that the current interfaces in their unmodified state are insufficient to openly and portably implement a transport plug-in. With no standard representation of a Tagged Profile ORB specific implementation is required to support marshalling and un-marshalling of Tagged Profiles.

On a more positive note, it is felt that the ETF style of interfaces have certainly provided PrismTech as a vendor with benefits. This work has made it possible to write a transport plug-in once and plug it into both the C and C++ implementations of our SCA-CORBA SDR embedded operating environment. It is also important we believe for any shipping pluggable transport SDK to provide source and examples required to implement the ORB specific protocol mentioned earlier.

Some of the most critical shortcomings are that

(1) The specification is at present more useful to an ORB vendor, or an end user, rather than a third party transport developer.

(2) At present a transport plug-in cannot be written once and plugged into two ORBs from different vendors.

(3) TheSeparation between message, protocol and transport layers is not enforced cleanly enough.

Some of these issues have been raised in the discussion part of this paper and should be addressed before the final specification is adopted. Finally, perhaps the most important issues in SCA development today revolve around standards compliance in order to achieve true waveform portability across all elements of the radio as far as possible. The work of this paper goes towards helping achieve that goal with the aim of achieving a simplification and componentization of the underlying transport connection abstraction in a portable manner also. This paper therefore builds upon our assertion from our 2004 paper in which we proposed the possibility of a GIOP-everywhere ideology inside the SCA radio. It shows that it is indeed possible to achieve a complete ubiquitous SCA machine across the radio. This unburdens the waveform developer from having to have intricate knowledge of the details of the s/w and h/w of the physical radios hetergenous interconnects, therefore yielding substantial cost and time benefits to the wireless industry's value chain.

REFERENCES

[1] Pugh K., *Prefactoring – Extreme Abstraction, Extreme Separation, Extreme Readability*, O'Reilly 2005.

[2] Mitola J., – *Software Radio Architecture – Object oriented Approaches to Wireless Systems Engineering*, Wiley 2000.

[3] Reed J.H., – *Software Radio, A modern approach to Radio Engineering*, Prentice Hall 2002.

[4] Dohse D., Bush L., Osborne G., Christensen E., –*"Successfully introducing CORBA into the signal processing chain of a software defined radio"*, COTS Journal, January 2003.

[5] Fette B., LaMacchia M , Christensen E.,– *"High Performance Software Radios"*, April 2004.

[6] The OMG *Extensible Transport Framework Specification* – OMG Document reference ptc/04-03-03.

PRISMTECH
Productivity Tools & Middleware

Practical Experiences using the OMG's Extensible Transport Framework (ETF) under a real-time CORBA ORB to Implement QoS Sensitive Custom Transports for SDR

Shahzad Aslam-Mir Ph.D.
Chief Technology Officer
PrismTech Corporation

Agenda

1. Background
2. ETF Interfaces
3. Transport Plug-in Case Studies
4. Practical Experiences
5. Conclusions

Background

- Objective – to establish a framework for plugging in transports in an ORB with sufficient predictability in order to support DRE systems
- WHY – IIOP (GIOP over TCP/IP) enables reliable remote messaging, however TCP/IP introduces unpredictable latencies unsuitable for many real-time systems

Background

- **Scope**
  - GIOP messaging and CDR encoding is adequate for real-time systems
  - No requirement to specify a alternative messaging protocol to GIOP
  - Should provide clear separation on concerns between the messaging layer (GIOP) and the transport layer (e.g TCP/IP)
  - Specifically by defining interfaces to enable ORB core, facility and service layers to be independent of the underlying transport
  - Facilitate the development of 3rd party transport solutions

Background

- **Key Requirements**
  - Must support an IOR architecture for non TCP transports such that it is possible for a transport author to create a transport plug-in for two different ORBs that enable application interoperability across the transport
  - Clearly identified interfaces and interaction semantics between the ORB and the plugin
  - How an ORB selects a transport should be specified

Background

- **Resulted in:**
  - Extensible Transport Framework Specification (document reference ptc/04-03-03)
  - Which is an OMG adopted specification
  - With submissions or contributions from the following companies:
    - Borland Software Corp.
    - Objective Interface Systems, Inc.
    - VERTEL Corp.
  - Current Status:
    - Undergoing finalization

ETF Interfaces

- A compliant plugin implements the following interfaces:
  - ETF::Profile
  - ETF::Connection
  - ETF::Listener
  - ETF::Factories
- An ORB compliant with the specification implements the following interface:
  - ETF::Handle
- Optional interfaces that the plugin can implement in order to conform to the “zero copy “ compliance point are:
  - ETF::ConnectionZeroCopy
  - ETF::BufferList

ETF Interfaces

- **ETF::Factories**
  - Local interface
  - Provides ‘entry point’ for ORB to use transport
  - Plugged into ORB via proprietary mechanism
  - Identified by IOP::ProfileId (IIOP etc)

  - create_listener(…) : ETF::Listener
  - create_connection(…) : ETF::Connection
  - demarshal_profile(…)

ETF Interfaces

- **ETF::Listener**
  - Local interface
  - Endpoint which clients contact when connecting
    - Associated ETF::Profile endpoint (its transport address)
  - Encapsulates Connection establishment protocol
    - May be provided by underlying transport (TCP)
    - Otherwise implemented in plugin code (SHMEM)
  - ORB may use blocking or non-blocking style
    - ORB thread calls blocking accept() operation
    - ETF thread calls ORB via ETF::Handle callback

ETF Interfaces

- **ETF::Connection**
  - Local interface
  - Encapsulates semantics of Connection protocol
    - Reliable, ordered, 1-to-1, bi-directional byte stream
  - Overloaded interface for client and server side
  - Initiated from client-side
    - ORB creates a Connection using Factories
    - ORB calls connect() to establish connection
  - Server-Side
    - Listener creates new Connection object in response to incoming request from client.
  - Client and server then read/write over Connection

ETF Interfaces

- **ETF::Profile**
  - Local interface
  - Encapsulates the conversion and matching functions used to store transport specific profile data in an IOR
  - Can also be used to locate a “matching” profile read from an IOR
  - Holds data related to an address for a transport
  - marshal() function creates an ETF::AddressProfile which packages all profile address data into an octet sequence

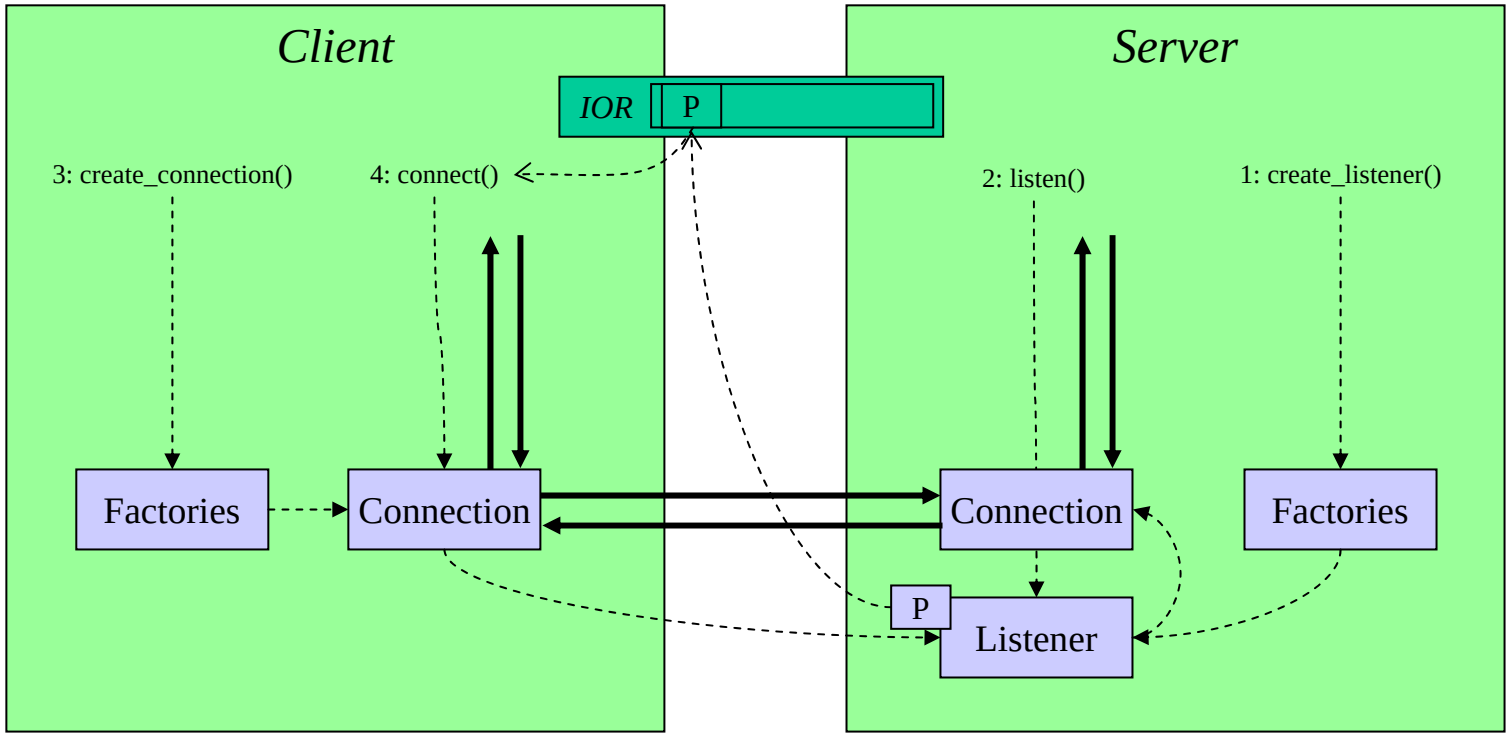
ETF Interfaces

- **ETF::Handle**
  - Local interface
  - Implemented by the ORB
  - ORB registers a Handle with ETF
  - Enables flexible threading models
  - ETF then makes up-calls to Handle when:
    - A new connection has been established
    - Data has arrived on an existing connection
  - ORB thus avoids some blocking calls to ETF.
  - ORB must still make some blocking calls:
    - connect() & write() still have to be blocking calls
  - Only available on the server side

ETF Interfaces

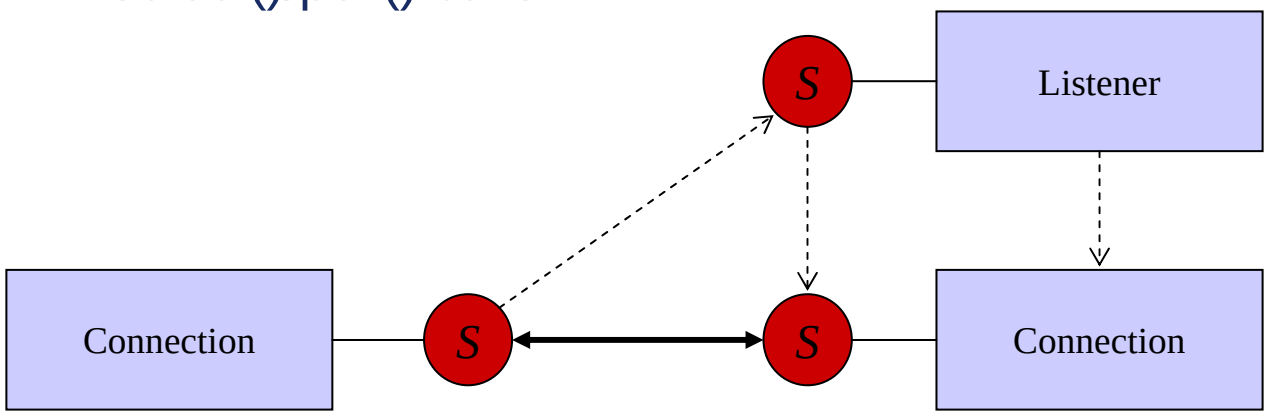
- **ETF::ConnectionZeroCopy : Connection**
  - Local interface
  - Optional compliance point within the standard
  - Supports the notion of a “zero copy” data transfer into the transport layer
  - Provides operations to write and read zero copy compatible buffers to and from the transport:
    - void write_zc(inoutBufferList data,…..
    - void read_zc(inoutBufferList data,…..
- ETF::BufferList is a local interface that provides operations that manage the allocation of a buffer compatible with the zero copy transport mechanism

ETF Connection Establishment



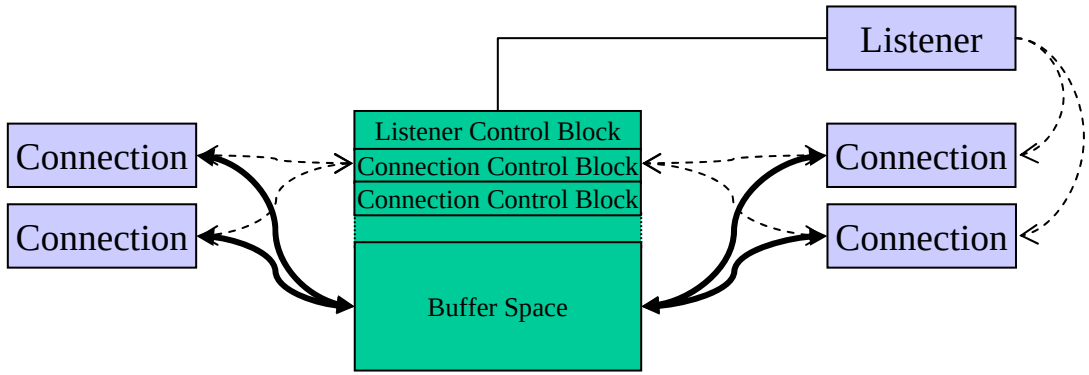
TCP Sockets Implementation

- ETF::Listener encapsulates a listening socket
- ETF::Connection encapsulates a connected socket
- ETF::Profile encapsulates an endpoint specified by host & port
- Hides details of sockets API
- Implements timeouts with:
  - non-blocking sockets
  - select()/poll() calls



Shared Memory Implementation

- ETF::Listener creates & manages shared memory segment
- ETF::Listener allocates a control block at start
- ETF::Profile encapsulates an endpoint specified by a file name
- ETF::Connections get allocated a control block each
- Remainder of shared memory segment used for transfer buffer – shared by connections
- Coordination by:
  - POSIX : Semaphores
  - System V : Message Queue



Experiences

- Benefits:
  - PrismTech have a single unified transport plugin API underneath our RTE ORBs (e*ORB SDR C and C++ Editions)
  - Both ORBs share the same pluggable transport layer code implemented once in C
  - Allows us to write a transport plug-in once in C and plug into either ORB without excessive porting
  - C++ based transport plugins can be also be plugged into the ORBs – however the C ORB has no way of handle any exceptions thrown by transport written in C++
  - The new ETF transport plugin interfaces are an improvement over the original proprietary interfaces – simpler abstraction, easier to document and explain to end users

Experiences

- Issues:
  - Confuses the boundaries between transport (e.g TCP), the messaging layer (e.g GIOP) and the protocol (e.g IIOP)
    - The spec states "separates the message layer (GIOP) from the Extensible Transport Framework"– but it doesn't
    - ETF::Profile includes a supported GIOP version attribute should be protocol version if required (e.g. IIOP::Version)
    - ETF::Profile::marshal() operation assumes CDR
    - ETF::Factories::demarshal_profile(in AddressProfile profile) operation requires ORB to understand AddressProfile encoding for each protocol

Experiences

- Issues:
  - The spec states "a Factories object needs to have an identifier so that the ORB can select the correct transport type"– but it doesn't.
    - An ETF::Factories object is identified by an IOP::ProfileId – much more than "transport type“
    - An IOP::ProfileId identifies a protocol,which implies a specific message layer and tagged profile encoding etc
    - Cannot use a transport plugin with different message layers to form different protocols.

Experiences

- Issues:
  - Issue 7594:ETF::Profile::marshal() and ETF::Factories::demarshal_profile() are unworkable:
    - The ETF plugin has no access to IOP::TaggedComponents in the full IOP::TaggedProfile. When marshaling a tagged profile,the ETF plugin may want to add tagged components, and when unmarshaling it may want to read them
    - When marshaling a full tagged profile,the ETF marshals part of it (ETF::AddressProfile), and then the ORB marshals the rest (IOP::ObjectKey & IOP::TaggedComponents). For IIOP, at least, these two parts must form a single encapsulation
    - The responsibility for encoding and decoding a full tagged profile is split between the ORB and ETF. The result is that neither party has access to all the information necessary

Experiences

- Issues:
  - ETF::Handle serves two distinct roles:
    1. connection establishment
    2. message arrival
  - It would be better to have two separate handle interfaces to encapsulate the different behaviour, this would enable:
    - call-backs and blocking for different areas e.g. call-backs for connection establishment and blocking calls for data transfer
  - Cannot use ETF::Handle interface on client side connections – when de-multiplexing replies from a shared connection, the ORB must use a dedicated I/O thread to do blocking/polling calls on the connection

Conclusions

- Provides a standard set interfaces by which a ORB transport plugin can implemented
- Currently specified interfaces are not enough to successfully implement a complete transport plugin, additional ORB level implementation is required per plugin
- More useful at present to an ORB vendor, or an end user, rather than a third party transport author
- Transport plugin cannot be written once and plugged into two ORBs from different vendors
- Separation between message, protocol and transport layers is not enforced cleanly enough
- Changes required during specification finalization to address fundamental issues