PrismTech
Distributed Software Infrastructure

Practical Experiences of using the OMG's Extensible Transport Framework (ETF) to Implement QoS Sensitive Custom Transports

Andrew Foster
Product Manager
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 specifies a transport should be specified

Background

- **Resulted in:**
  - Extensible Transport Framework Specification (document reference ptc/04-03-03)
  - Which is an OMG final 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

- **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

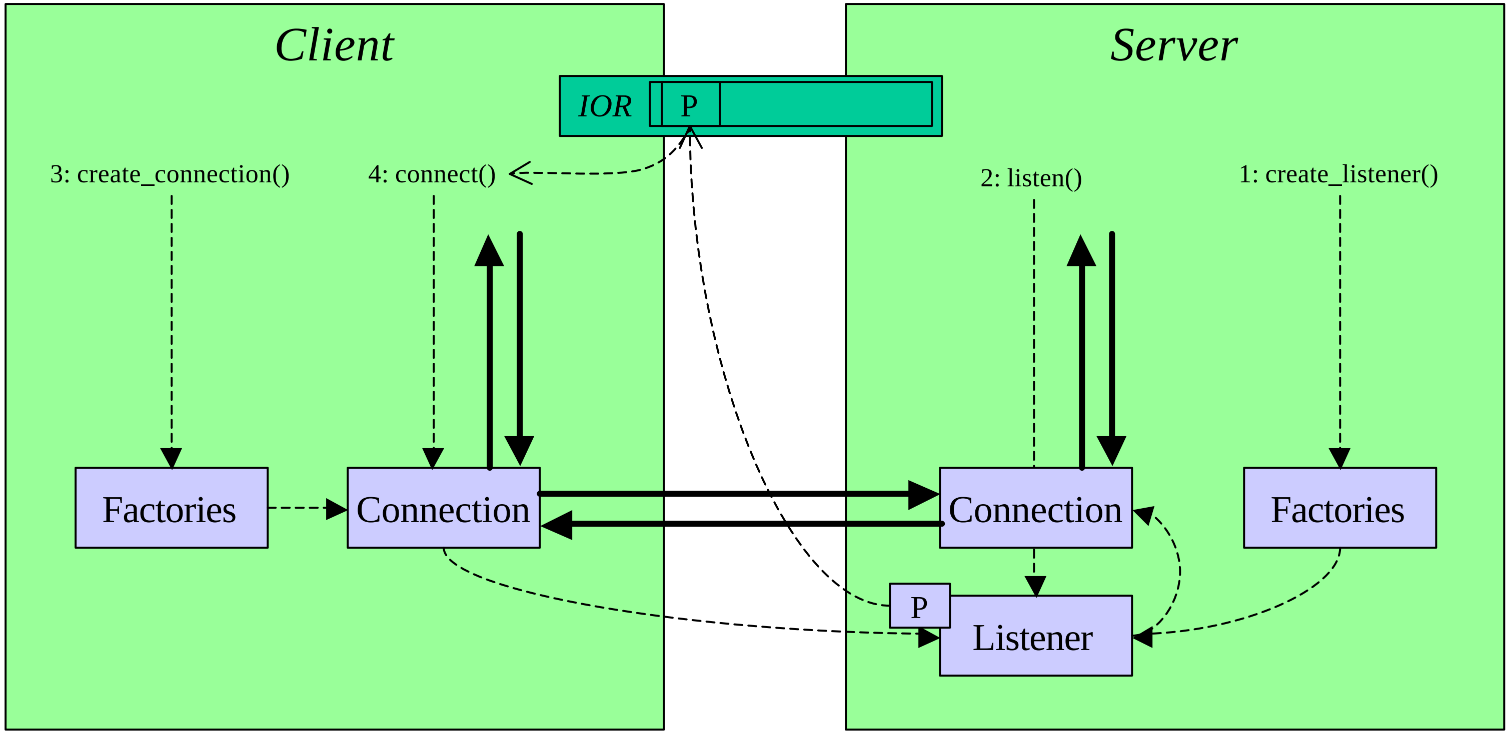
ETF Interfaces

- **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

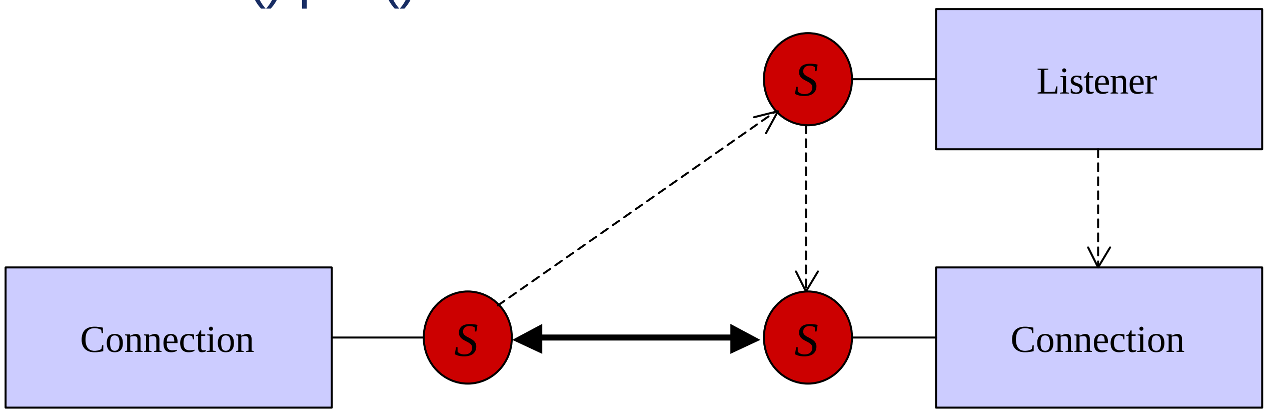
- **ETF::ConnectionZeroCopy : Connection**
  - Local interface
  - Optional compliance point within the standard
  - Supports the notion of a “zero copy” data transfer into the transport layer
  - Provides operations to write and read zero copy compatible buffers to and from the transport:
    - void 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



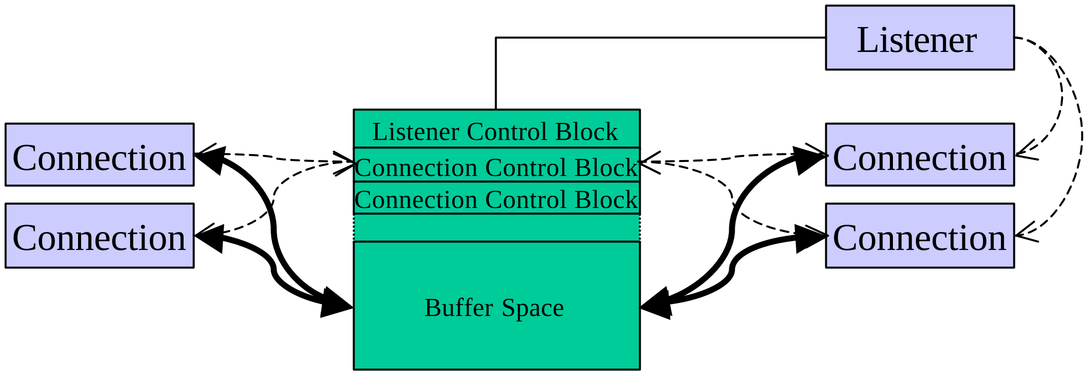
TCP Sockets Implementation

- 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() operation assumes CDR

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

