Multiple Transport Implementations

Strategies for Increased Configurability Using the FACE™ Technical Standard, Edition 3.1

The Open Group FACE™ Army TIM Paper by:

Christopher J. Edwards, Systems Engineering Lead, CMS Team

Steven P. Price, Software Engineer, CMS Team/FACE TSS SC Lead

Rachel Moudy, Software Engineering Lead, CMS Team

Shaun Foley, Senior Software Engineer, Skayl

May 26, 2021

Table of Contents

Executive Summary

Reuse of software is a business objective for the Department of Defense (DoD) as a mechanism to reduce costs for software related expenses. The effort to reuse software is directly related to the design of that software and the target platform. Through common use of the FACE™ Technical Standard, software architectures would be aligned, reducing the impact of porting capability software from one platform to another.

The system integrator is responsible for much of the porting and reusing as directed by the platform. The variability in the development approach of software conformant to the FACE Technical Standard can impact the effort a system integrator has to incorporate that UoC into an existing system.

Several methods to approaching the integration of Transport Services Segment (TSS) interfaces to Units of Conformance (UoCs) that use those services are presented here. The results of this examination and recommendations are presented in this paper.

Background

The Rapid Integration Framework (RIF) is a set of components originally developed to the UH-60M Crew Mission Station (CMS). The FACE approach was used in the development of the CMS Project (Edwards, Price, & Mooradian, The Impact of the FACE Technical Standard on Achieving the Crew Mission Station (CMS) Objectives, October, 2017). This use allowed the CMS components to be used in integration demonstrations proving the value of the FACE approach at the FACE TIM of 2018 (Edwards, Rapid Integration Framework (RIF) Demonstration Information Packet, 2018).

As part of the 2018 integrations, Skayl demonstrated integration of software components without recompile using a data model integration approach. In 2021, RIF components were used to integrate alternative TSS solutions. Components were converted to the FACE 3.1 Technical Standard. Experiments included both using alternative Transport Service Segment (TSS) implementations and the use of multiple TSS implementations within the same system. Alternative TSS solutions integrated into the CMS include the Skayl product along with other TSS implementations.

One of the principal goals of the RIF experiments is to reduce the efforts to bring new capabilities to the field. The FACE approach is aimed principally at software reuse, focusing on portable/reusable software as a means to reduce software costs in the DoD.

If the goals of FACE Conformance are realized, software Units of Conformance (UoCs) that provide a functional capability would not change as the logic is ported/reused on new platforms. The effort of bringing a capability to a new platform will primarily fall to the system integrator. The principal efforts would include the integration of that capability into the system via the TSS software.

This paper introduces some best practices for implementation into UoCs in the TSS, PCS, and PSSS segments. When implemented, these best practices make the job of the system integrator easier and will reduce overall integration time. This paper also addresses strategies for the system integrator when these best practices are not followed in the development of software being integrated.

Note on the use of UoC: This paper is focused on the integrator’s effort to integrate multiple UoCs into a system. These UoCs are typically in the Portable Component or Platform Specific Services segments, particularly when describing how Transport Services are used. There are also cases where a UoC refers to software in the TSS. The system integrator is responsible for integration of TSS components as well as integrating other components to the TSS. Most cases the term UoC will include the name of the interface that UoC uses in order to give context. In cases where UoC is used without a segment or interface, the text refers to UoCs from any of the three segments.

Multiple Transport Implementations

Transport Services Capabilities

The FACE Technical Standard, Edition 3.0 (FACE Consortium, December 2017) and later offer several capabilities that are analyzed as part of this paper.

Transforms

Data Transformations include the ability to transform a single data element from one Data Modeled type to another. This can consist of unit conversion, synchronizing enumerations, or conversion from basic types (i.e., integer to float). Additionally, it includes combining data from multiple sources into a new message and sending the data at a different rate than the source. Data Transformation will be a regular aspect of integrating a conformant UoC from a different system without changing that UoC. Data Transforms are also an essential part of supporting configurable core capabilities (Edwards, Price, & Tanner, Transformation Capabilities in Configurable Common Services, 2018).

When applying this to multiple TSS implementations, the transform approach should factor in the receipt of data from differing TSS implementations.

Transport Protocol Module (TPM)

The TPM provides a means of linking two types of transports together, allowing the use of multiple transport mechanisms within the logic of the Transport Service Segment (TSS).

Type abstraction

The Type Abstraction interface is provided to simplify conformance and separate the type specific aspects of a TSS from the basic transport of the data. A TSS UoC can be implemented to a Type Abstraction interface. It can then go through FACE Conformance without the need to add code to accept new types as they are added to the system. A TS-TA Interface Adapter can be added to this TSS for each new type without impacting the conformance of the larger TSS.

Therefore, a UoC in the TSS can provide the Type Abstraction interface and maintain conformance as new types are added through TS-TA Interface Adapter. In a previous paper (Edwards, Price, & Tanner, Transformation Capabilities in Configurable Common Services, 2018), the RIF team proposed that the Data Marshalling and Transform capabilities are best suited for implementation in the TS-TA Interface Adapter to minimize the areas that are type aware.

Code generation of new data types

Most transport products offer a means to generate the Type Specific interface. This can typically be from a FACE Technical Standard Data Model or through header files generated from the Conformance Test Suite. The FACE Conformance Program allows a certified conformant TSS UoC to provide a process for adding types without running through conformance with the newly generated software, this allows maintaining conformance as types are added.

Multiple Transport Implementations

System Integrator Responsibilities

When a system integrator accepts a new PCS or PSSS UoC into a system, it should be intended not to change that UoC's code. When using FACE Technical Standard, Edition 3.0 or later, some integration software must be developed to link the Transport UoCs with the PCS and PSSS UoCs.

Configure the TSS Message and Data Flows

An obvious function provided by the system integrator is managing the data flows from one PCS/PSSS UoC to the next. This routing of data is accomplished through TSS Configuration. System integrator knowledge of how the TSS is configured and managed greatly influences the success of a timely integration effort.

Tooling provided with a TSS Solution, such as modeling tools to graphically connect the FACE Technical Standard required UoC Supplied Models (USMs), can ease the effort in configuring a TSS. These tools can also aid in identifying areas where transforms are needed.

New types from a data model

One of the primary things a system integrator must do is adapt the system to use the new data types from a new UoC integration. Most PCS/PSSS UoCs' integrations will extend the number of messages and data types that the TSS must support. Therefore, it will require adapting the TSS to support these new messages and/or data types through the mechanism presented by the TSS.

The UoC provider may provide a TS-TA Interface Adapter for the Type Abstraction Interface that eliminates this work along with the UoC. Care must be taken to integrate messages into the system properly. Data Marshalling and Transform work may also be needed.

Recommended Practice:

Transport solutions should provide tooling to support system integrator tasks such as adding new types and configuring the system data flows.

Injectable Interface

Per the FACE Technical Standard, concrete instances of the TSS interfaces are provided through an Injectable Interface. This means that the UoC using the transport services will provide a function for accepting the implementation of TSS interfaces. This injectable method allows a single linked executable[1] to support multiple implementations of interfaces. The system integrator code will include instantiations of each UoC in the executable. The integration code will pass the references of the correct interface instantiation to the instantiations of the UoCs that use those interfaces.

[1] The term executable in this document refers to the combined set of libraries and source code that is linked together into a single image (in ARINC 653) or executable (in POSIX).

Multiple Transport Implementations

TSS Type Specific and its Injectable

The TSS Type Specific interface provides read and write functionalities of the connection. The TSS Interface requires a unique TSS interface for each data type used by each PSSS and PCS UoC.

Some PCS/PSSS UoCs will support multiple connections for individual data types. A UoC may be designed to use a single Set_Reference for all of these connections using a common data type. However, it could also be designed to need a separate TSS Set_Reference call for each connection. The system integrator may desire that some of these connections come from one TSS implementation source while others come from a different TSS implementation.

Recommended Practice:

UoCs using the Type Specific Interface should provide a Set Reference interface for each connection not just per type.

TSS Base and its Injectable

The TSS Base interface provides the Initialize, Create_Connection and Destroy_Connection methods. When a UoC calls the Create_Connection call, the system integrator code must ensure that the Create_Connection for the TSS Base implementation comes from the same TSS product the connection will use.

Supporting Multiple Transport Implementations

A transport implementation offers a wide variety of services to the system it supports. Services can range from providing connectivity across multiple systems to providing connectivity to two PCS/PSSS UoCs within the same address space. Communication may also use Pub/Sub or Command/Response mechanisms. A wide variety of connected systems may use varying means to place data on the wire or even use differing wiring hardware. Communication between two UoCs in the same address space may be most efficient if using shared memory and read-callback functions, which some TSS implementations may not support.

Implementation of a single approach to the basic transport will lead to tradeoffs in complexity versus effectiveness. Planning for a flexible transport implementation provides a means to support the breadth of TSS possibilities while allowing a simple solution to the needs of each data flow. A single TSS implementation that supports all capabilities may not be needed on a system that only uses Pub/Sub. A system built with only Pub/Sub capabilities may have extra work ahead to add a connection to a Common Object Request Broker Architecture (CORBA) based implementation on another platform. One task for the system integrator is to determine the capabilities needed for the TSS.

As PCS/PSSS UoCs are integrated into the system, new TSS capabilities may be required. Additionally, upgrades to the TSS functions may occur. As the TSS is linked to every application within the system, modification of the TSS could have far-reaching implications for testing.

The use of a multiple TSS approaches in system design can mitigate future rework and qualification costs. During S3I's work to test new TSS implementations, only some of the messages were moved to a new TSS implementation. UoCs using messages from multiple TSS implementations were configured to use the new messages on the new TSS while existing messages remained on the original TSS. The result was that only a small number of the UoCs in the system were affected by the new TSS.

Support of Multiple Editions of the FACE Technical Standard

The support of multiple editions of the FACE Technical Standard follows a similar need. It is desirable for a system to support UoCs written to FACE Technical Standard, Edition 2.1 and still support UoCs conformant to Edition 3.0 and future versions of the technical standard.

As part of the FACE Expo in 2018, the compatibility between the FACE Technical Standard, Editions 2.1 and 3.0 was demonstrated by using a transform within the TSS to convert Basic Avionics Lightweight Source Archetype (BALSA) messages (Edwards, Rapid Integration Framework (RIF) Demonstration Information Packet, 2018).

Recommended Practice:

Ensure transport solutions provide a wire mechanism that supports multiple editions of the FACE Technical Standard and implement a plan for maintaining that support.

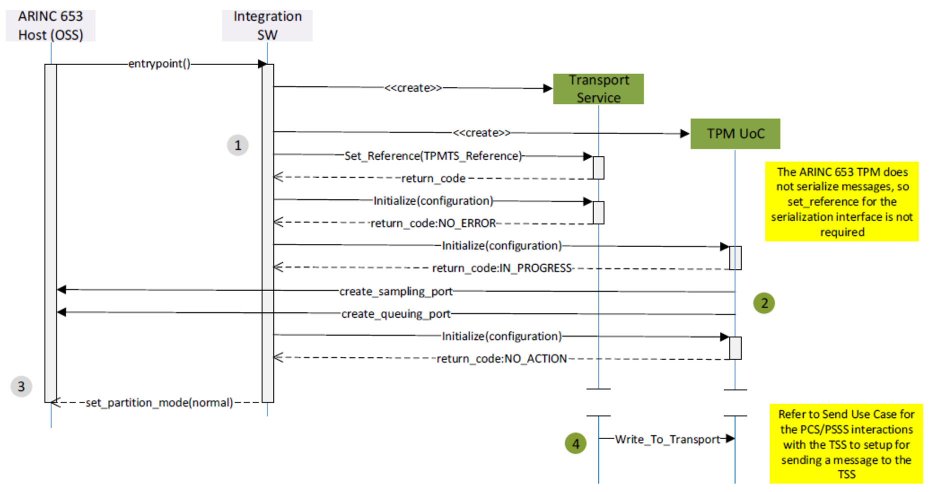
In 2020, the RIF Team began converting UoCs over to FACE Technical Standard, Edition 3.1 using a modification of the TSS and using a data model conversion that provided the same bit-wise messages. As each UoC was converted, it was tested against the other FACE Technical Standard, Edition 2.1 components.

Multiple Transport Implementations

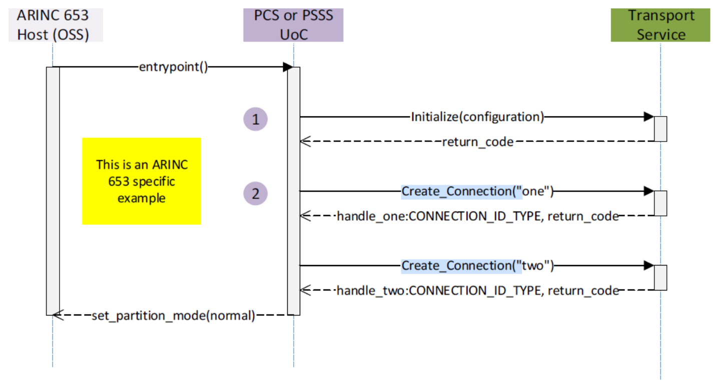
Integration Using the TPM

The Transport Protocol Module (TPM) interface defined in the FACE Technical Standard, Edition 3.x allows two TSS implementations to connect (FACE Consortium, May 2020). This approach has the interface to the PCS/PSSS UoC using one TSS implementation. The TPM is best seen as a bridge between two systems using different transport mechanisms. TSS UoCs can be designed to use a TPM interface to take advantage of these benefits.

Implementation Details



ARINC 653 Host (OSS)
Integration SW
entrypoint()
<<create>>
Transport Service
<<create>>
TPM UoC
1
Set_Reference(TPMTS_Reference)
return_code
Initialize(configuration)
return_code:NO_ERROR
Initialize(configuration)
return_code:IN_PROGRESS
The ARINC 653 TPM does not serialize messages, so set_reference for the serialization interface is not required
create_sampling_port
create_queuing_port
2
Initialize(configuration)
return_code:NO_ACTION
3
set_partition_mode(normal)
4
Write_To_Transport
Refer to Send Use Case for the PCS/PSSS interactions with the TSS to setup for sending a message to the TSS



ARINC 653 Host (OSS)
PCS or PSSS UoC
Transport Service
entrypoint()
1
Initialize(configuration)
return_code
This is an ARINC 653 specific example
2
Create_Connection("one")
handle_one:CONNECTION_ID_TYPE, return_code
Create_Connection("two")
handle_two:CONNECTION_ID_TYPE, return_code
set_partition_mode(normal)

Figure 1-Initializing a TPM Implementation

When the TPM Set_Reference interface is invoked it passes the instance of the TPM UoC that was created. The Transport Service will be responsible for retrieving the configuration and initializing the parameters it requires. The system integrator will be responsible for setting the configuration parameters appropriately for the reference they need to pass.

Additionally, the use of serialization methods is beneficial for retrieving instances of each message interface. Those messages will have a unique identifier associated with it to help ensure the instance that is passed can be marshalled or unmarshalled based upon the Interface Definition Language (IDL) data type.

Multiple Transport Implementations

System Integrator Effort

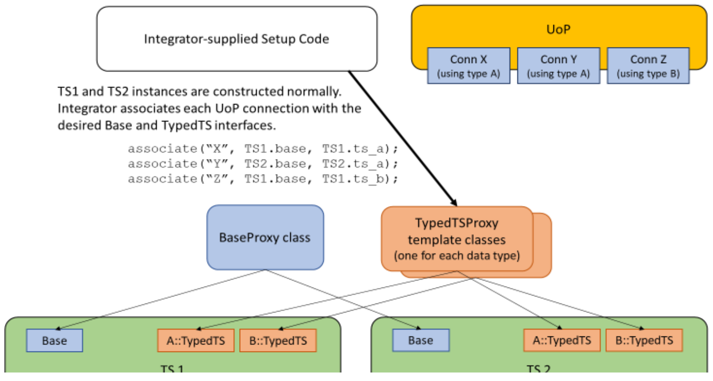
For each of these approaches we will be examining the same five questions see Table 1: Using TPM Base Details. Error handling is omitted, to focus on the more fundamental differences in the approaches.

Table 1: Using TPM Base Details

Category	**Details**
What interfaces are injected into various related UoCs?	Type Specific Base Type Specific Typed (per type) TPM Interface (per TPM)
How many steps in the Read/Write data flows?	Includes marshalling calls
How many functions does the system integrator write?	Setting all the References Custom Serialization (possibly)
How many injectable interface calls?	1+ [number of TSS BASE] + [number of TypeSpecific]
Is the system integrator additionally burdened with new types, marshalling, transforms?	Custom serialization code can simplify the transfer of messages between protocols

Integration Using Custom TSS Base and TypedTS Proxies

One mechanism for integrating two TSS implementations is to create proxies for the TS-related interfaces injected into the PCS or PSSS UoC. For this proof-of-concept, a PCS/PSSS UoC originally using one TSS implementation was integrated to use two TSS instances. Previously, the TSS::Base and TSS::<data-type>::TypedTS provided by a single Transport Service were directly injected into the UoC. Neither the UoC nor the TSS implementation were changed; all modifications were to system integrator -supplied “main()” code.


Integrator-supplied Setup Code
UoP
Conn X (using type A)
Conn Y (using type A)
Conn Z (using type B)
TS1 and TS2 instances are constructed normally. Integrator associates each UoP connection with the desired Base and TypedTS interfaces.
BaseProxy class
TypedTSProxy template classes (one for each data type)
Base
A::TypedTS
B::TypedTS
TS 1
Base
A::TypedTS
B::TypedTS
TS 2

```
associate(“X”, TS1.base, TS1.ts_a);
associate(“Y”, TS2.base, TS2.ts_a);
associate(“Z”, TS1.base, TS1.ts_b);
```

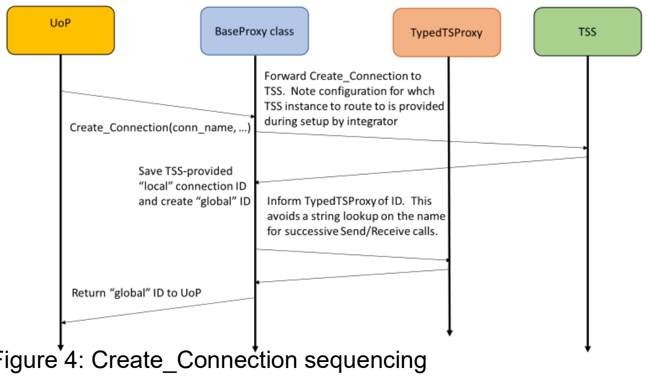
Figure 2 -Interfaces for Proxy Base and TypedTS interfaces

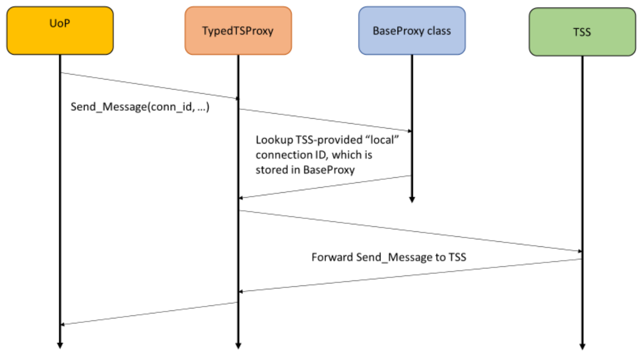
While another layer (albeit a thin one) is not ideal, it may be necessary in cases where the UoC does not support being injected with multiple TS Bases or TypedTS instances. The UoC in this exercise expected to be injected with a single Base (in contrast to per-TypedTS) and a TypedTS per-type (in contrast to per-connection), but a similar approach will likely work for UoCs with different injectable expectations.

Multiple Transport Implementations

Implementation Details

Two new classes were implemented: a BaseProxy class and a TypedTSProxy template class. BaseProxy holds a reference to the TSS::Base provided by each TSS instance, and TypedTSProxy holds a reference to the <data-type>::TypedTS provided by each TSS instance. Only a single BaseProxy object is created, while a separate TypedTSProxy is required for each data type, since its APIs are type-specific. These proxy instances are injected via Set_Reference into the UoC. After creating the separate TSS instances exactly as they would be independently, the system integrator configures the proxies so that UoC function calls are routed to the appropriate TSS instance.


UoP
BaseProxy class
TypedTSProxy
TSS
Forward Create_Connection to TSS. Note configuration for whch TSS instance to route to is provided during setup by integrator
Create_Connection(conn_name, ...)
Save TSS-provided "local" connection ID and create "global" ID
Inform TypedTSProxy of ID. This avoids a string lookup on the name for successive Send/Receive calls.
Return "global" ID to UoP
Figure 4: Create_Connection sequencing


UoP
TypedTSProxy
BaseProxy class
TSS
Send_Message(conn_id, ...)
Lookup TSS-provided "local" connection ID, which is stored in BaseProxy
Forward Send_Message to TSS
Figure 3 - Send_Message sequencing

There were two main complications to implement this routing. First, because each TSS instance is only responsible for providing a unique connection ID within its own scope, BaseProxy::Create_Connection cannot blindly return the ID. Instead, it must return an ID that is unique across all TSS instances and maintain a mapping from this "global" ID to the "local" TSS-provided ID.

Second, and related, TypedTSProxy and BaseProxy must reference each other, which is somewhat awkward. This is because TypedTSProxy is given a "global" ID, but must provide a "local" ID to the underlying TSS instance. Furthermore, the routing to a TSS must be configured by connection name, not by ID – but the BaseProxy implements Create_Connection where this linkage between connection name, "global" ID, and "local" ID is made. Thus, TypedTSProxy must get this linkage from BaseProxy.

Note that only the Send_Message path was implemented for this prototype; however, the pattern should apply for all TypedTS APIs

Multiple Transport Implementations

Lessons learned from the implementation

The Proxy approach looks at “impedance mismatches” between how the UoC expects the Base and TypedTS interfaces to be injected and used. The separation of functions into separate Base and TypedTS interfaces adds a lot of incidental complexity. An informal proposal for simplified APIs is in-progress. The core simplification is to remove all or most Base methods and to require that a TypedTS reference be injected for each UoC Connection. The interface_name parameter to Set_Reference would match the Connection name. Thus, there is no need for UoCs to maintain the connection_id returned by Create_Connection – the reference itself would uniquely identify the Connection. Then the system integrator would not need to create proxy classes and could merely inject the desired TypedTS interface (i.e. the one provided by the TS instance the system integrator wishes to associate with the connection) into the UoC.

Integration Using Custom TSS Base and Direct Type Specific Calls

In a variation of this approach, it is also possible to simply set the Type Specific references to the TSS implementation that will be used for the connection. In this case the Custom TSS Base would simply return the correct Connection ID from the create connection call.

A simpler mechanism for integrating two TSS implementations may be to allow the Type Specific API calls made by the UoC to go directly to the underlying TSS implementation. In this case the call to Create Connection must be routed to the correct TSS implementation. One means of handling that is to use a custom TSS Base configured to pass the Create Connection on to the proper TSS Base.

This implementation should work no matter how the UoC decided to implement the TSS Set_Reference calls. As it takes advantage of the separation from the TSS Base and the Type Specific implementations.

System Integrator Effort

For each of these approaches we will be examining the same five questions see Table 2: Using a Custom TSS Base Details. Error handling is omitted, to focus on the more fundamental differences in the approaches.

Table 2: Using a Custom TSS Base Details

Catagory	**Details**
What interfaces are injected?	Type Specific Base Type Specific Typed (per type)
How many steps in the Read/Write data flows?	Two integer lookups, then forwarded to TSS instance call
How many steps in the Create/Destroy connection data flows?	Two string lookups, two integer lookups, after TSS instance call returns
How many functions does the system integrator write?	One for each function in TSS::Base and TypedTS interfaces, plus six for BaseProxy class and TypedTSProxy template classes.
How many injectable interface calls?	1 + [# of Data Types]
Is the system integrator additionally burdened with new types, marshalling, transforms?	No, but may depend on how TSS is implemented

Multiple Transport Implementations

Integration Using Multiple TSS Bases

Another mechanism to allow the Type Specific API calls made to by the PCS/PSSS UoC to go directly to the underlying TSS implementation is to pass each TSS Base to the UoC, so the UoC calls the correct Create_Connection call for each connection. This is similar to the use of a Custom TSS Base separate from the Type Specific, but also reflects the forward-looking approach to combine the TSS Base and Type Specific.

Recommended Practice:

UoC implementations supporting injections of multiple TSS Base instances support the greatest flexibility in TSS support.

Implementation Details

This implementation requires the UoC using the Type Specific interface to implement a mapping of the TSS Base to each Type Specific message. This requires a configuration parameter to identify the TSS Base for each connection. The TSS Base identifier passed into the TSS Base Set Reference can provide the value for this configuration item.

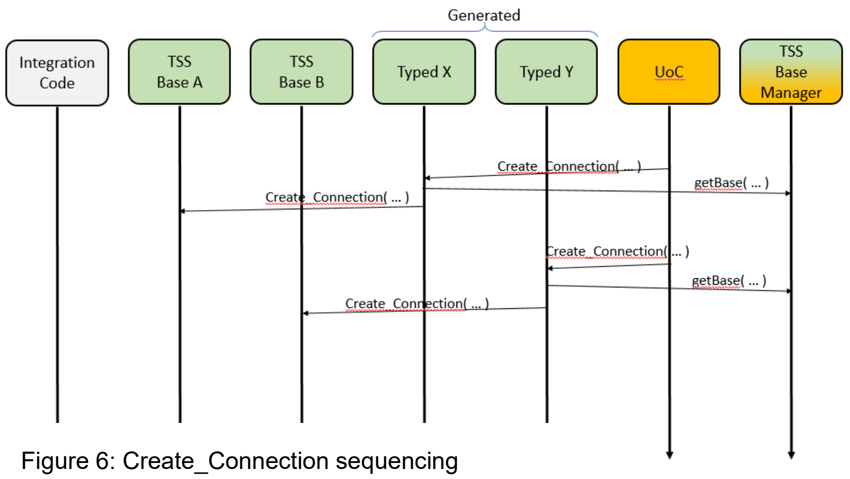


Figure 6: Create_Connection sequencing

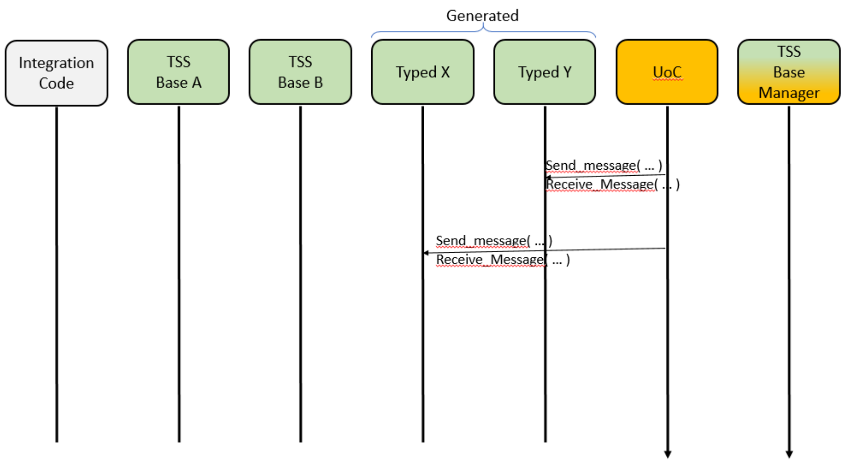


Figure 5 - Send_Message sequencing

Multiple Transport Implementations

System Integrator Effort

For each of these approaches we will be examining the same five questions see Table 3: Using multiple TSS Base Details. Error handling is omitted, to focus on the more fundamental differences in the approaches.

Table 3: Using multiple TSS Base Details

Catagory	**Details**
What interfaces are injected?	Type Specific Base (per TSS) Type Specific Typed (per type)
How many steps in the Read/Write data flows?	One for each
How many steps in the Create/Destroy connection data flows?	Three steps, a Create_Connection is sent to the Typed_Interface, which calls TSS_Base_Manager's getBase to get the correct TSS Base using the connection name, then Create_Connection is called referencing the correct TSS_Base.
How many functions does the system integrator write?	A function that performs Set_Reference calls
How many injectable interface calls?	[number of TSS BASE] + [number of TypeSpecific]
Is the system integrator additionally burdened with new types, marshalling, transforms?	No, however configuration of the UoC to allow for each Typed_Message to be assoicated with the correct TSS_Base.

The configuration of different TSS Base implementations generally requires unique resources. Within the configuration of any given PCS/PSSS UoC is an indication of where each injected Base can find its resource, based its name. That same name is also used for each typed connection, associating the Typed Message with a specific Base. This information is passed to the Base in the Initialize(…) call. The BaseManager initializes all the Base's that the integration code has injected into it. This is done by the name in the integration code matching the name of the Base in the UoC's configuration; both managed by the System Integrator. The BaseManager supports the Set_Reference, and holds a reference and the name of each Base.

The BaseManager class must be implemented in the PCS or PSSS UoC using the Transport Service interfaces. Use of this technique may mean placing the requirement on the UoC vendor.

Multiple Transport Implementations

Integration Tasks Related to Type Abstraction

The use of a Type Abstraction UoC is a factor the system integrator may consider when selecting a TSS. The use of a Type Abstraction UoC separates the core TSS functions from the type specifics, which can have advantages in the qualification of the TSS libraries, reducing the size of the software units that would require airworthiness analysis.

The design properties of a particular Transport Service implementation – for example, where capabilities such as serialization or transforms reside – may affect how easy it is to [re-]integrate but note that the PCS or PSSS UoC interaction will be largely unaffected. This is because there is no direct coupling of the TypeAbstraction UoC to PCS/PSSS UoCs. UoCs get references to <data-type>::TypedTS and Base, but no TypeAbstraction reference is injected. (Other references, such as FACE::Configuration may also be injected, but these are separate from Transport Service APIs). Thus, approaches that forward or proxy TypedTS and Base APIs should work whether a TS implementation chooses to use a TypeAbstraction UoC or not.

Another theoretical option to support using multiple transport implementations, assuming they are provided as separate TS-TA Interface Adapter and Type-Abstract UoCs, is to choose a single Type Abstraction implementation that is provided to both TS-TA UoCs. However, informal conversations with Transport Service implementers hint that there may be too many open questions regarding the intra-TSS-APIs to make this practical: Is data crossing the TA boundary serialized or just an opaque pointer to typed data? How is memory for data objects managed? Can the TS-TA UoC be implemented in a different language from the type-specific UoC?

Furthermore, it is unclear if there is a business need for this use case. The Transport Service Interfaces presented to PCS or PSSS UoCs are crucial for portability. The TPM API is necessary for cross-TS-domain interoperability. The TypeAbstraction API, on the other hand, benefits an implementation by separating typed and untyped code (thus easing conformance, code reuse, and safety-certification). That said, a system integrator could possibly write a similar forwarding/proxy class for the TypeAbstraction APIs to route function calls between separate TS-TA Interface Adapter UoCs and a single Type Abstraction UoC. Questions about data representation and memory allocation would need to be solved another way.

Multiple Transport Implementations

Integration Tasks Related to Transforms

Consider a case where a system integrator has a Transport Service that supports a known, fixed data type, but one that the PCS/PSSS UoC. How can these be integrated? The ideal answer is within the Transport Service via a Data Transform Capability, but what if that is not possible? Perhaps the Transport Service does not support transforms or is provided in binary form and is not editable[2]. A possible approach is discussed below, though it has not been prototyped.

Continuing the theme of integration "glue" code, such a transform could be performed in another proxy class that implements the <data-type>::TypedTS interfaces, intercepting the PCS/PSSS UoC's Send_Message() call, performing the data transform, and then forwarding to the original Transport Service's Send_Message(). The system integrator would need to provide, or at least have access to, the language-specific data type definitions and TypedTS interfaces; however, this is likely already available since the UoC is already using that type. Notably, the system integrator does *not* need to provide Serialization interfaces for the data type, since the original TS implementation never sees the new type.

Having covered the simplest case, things quickly become more complex from here. What if there are two Transport Service implementations with types A and B, but the PCS/PSSS UoC sends type C? A transforms interaction with more than two types becomes much more complex. A linear pipeline where a transform has a single input type and a single output type is convenient because the "in" and "out" APIs are analogous. The system integrator supplied transform code may need to handle conditions like the underlying Transport Service call blocking or returning an error, but the UoC code *already* had to handle the same conditions, so much of this handling can be punted up a level to the caller.

However, when the intermediate transform code calls *two or more* downstream functions, there are multiple approaches to handling errors. Understanding the sequence and semantics for using the Transport Service APIs are important in maintaining proper system state. Suppose the PCS/PSSS UoC sends with a timeout, the straightforward implementation would call Send_Message() for A and B in sequence, with appropriate timeouts calculated. An approach that spawns two threads concurrently requires pre-planned coordination. A producer-consumer queue is a common technique, but this adds another level of timeouts and buffering.

The receiving path is even harder since it must deal with both synchronous (Receive_Message) and asynchronous (callbacks) receipt styles. Furthermore, code that splits one type into two is guaranteed to get both types, or at least know at call-time there was a problem. Code that joins two types into one must deal with delayed or missing data which complicates correlation.

In summary, the FACE Transport Service Interfaces allow the system integrator to insert custom transform code between ported capabilities and the transport service without modifying either UoC. The system integrator however, is responsible for selecting and implementing an appropriate approach for error handling.

[2] Binary compatibility concerns regarding String, Sequence, and Fixed classes are out of scope for this paper.

Multiple Transport Implementations

Integration Tasks Using Code Generation

Code generation tools can greatly reduce the costs of integration. Integration tasks are greatly reduced when a TSS UoC provides for code generation to support new data types. Most TSS vendors provide a means to generate software from types expressed in the FACE USM.

The use of tools for system modeling and generation of TSS configurations from these system models are another instance of tooling that can greatly reduce system integrator work. Such tooling can abstract the details of TSS configuration from the user, allowing configuration through modeling tools.

The development of Data Transforms related to the integration of components will be of great use to a system integrator. As demonstrated by Skayl in 2018, the use of a System Model in addition to the USM can assist in the development of integration techniques without modification of application code.

The FACE Technical Standard, Edition 3.0 introduced more capabilities for the TSS that could be generated; some of these features may also be supplied by Component Frameworks or Operating Systems Services.

The procurement of a TSS implementation should consider the tooling supported by the TSS in generating integration software. Areas to look for include:

- Generation of Types from the USM, including the Type Specific Interface
- Generation of Data Transforms, including the combination of values from multiple messages, the transformation for units and basic types
- Generation of message specific serialization functions for marshalling through a TPM

Note: Generation of FACE data types may allow for advantages in the compile and link step over the use of the Type Abstraction UoC. The use of an Injectable Interface between the TS-TA Interface Adapter UoC and the Type Abstraction UoC prevents the compiler from taking advantage of optimizations that can reduce code size and eliminate some function call overheads.

Multiple Transport Implementations

Conclusion

We are all striving to stretch limited funding to acquire/produce more capabilities. Program Executive Offices (PEOs), Project Offices and Product Offices program offices direct the requirements placed on procured capabilities, including those enabled by FACE Conformant software. Selection of strategies in the implementation of transport services within a system should be reflected in the procurement of UoCs in the TSS, PSSS, and PCS to reduce the efforts required to integrate the software. Strategies should also include automating the integration, reducing the amount of software required for airworthiness, and ensuring the integration is well understood.

The approaches analyzed in the paper can all be used within system implementations when the UoC supports the capabilities. Summaries of the results are in Table 4: Approach Analysis.

Table 4: Approach Analysis

Method	**Requirements**	**Advantages**	**Disadvantages**
Integration Using TPM	Requires TPM development Should serialize messages to reduce overhead	Does not require recompile of existing software	Introduces overhead in the sending and receiveing of each message.
Use of a Custom TSS base	Requirement for support in the TSS implementation, can be developed by the system integrator. No requirements on the PSSS/PCS	Can be used with any UoC meeting the FACE Technical Standard Implements a mechanism that is forward looking, and may reduce rework for the next edition of the FACE Technical Standard Direct Type Specific call may provide the least overhead in the Send/Recieve Message	System integrator needs to write or generate, a TSS tool can mitigate.
Use of multiple TSS bases	Requires support in the PSSS/PCS UoCs	Features the least overhead in the Send/Recieve Message	UoCs may not support this requireing use of another method.

When analyzing the use of the TS-TA Interface Adapter v/s Code Generation without an abstraction interface, the use of the abstraction interface adds another injectable call that cannot be optimized out. This can lead to less efficient code. The use of this abstraction interface can, however, isolate the software into discrete libraries that can have full airworthiness artifacts, limiting the new type code to the smaller library that provides the Type Specific interface. The use of code generation that does not use the abstraction interface would be favorable in lower criticality software.

Having the flexibility to use the most beneficial method at the appropriate time influences procuring TSS UoCs and related tools.

Multiple Transport Implementations

References

(Please note that the links below are good at the time of writing but cannot be guaranteed for the future.)

Christopher J. Edwards, S. P. (2018). *Transformation Capabilities in Configurable Common Services.* The Open Group.

Edwards, C. J. (2018). Rapid Integration Framework (RIF) Demonstration Information Packet. *Proceedings of the 2018 September US Army FACE™ Technical Interchange Meeting.* Huntsville, AL: The Open Group.

Edwards, C. J., Price, S. P., & Mooradian, D. H. (October, 2017). *The Impact of the FACE Technical Standard on Achieving the Crew Mission Station (CMS) Objectives.* The Open Group.

Edwards, C. J., Price, S. P., & Tanner, W. G. (2018). *Transformation Capabilities in Configurable Common Services.* The Open Group.

FACE Consortium. (24 Jun 2014). *Technical Standard for Future Airborne Capability Environment (FACE), Edition 2.1.* The Open Group. Retrieved from www.opengroup.org/library/c145

FACE Consortium. (December 2017). *Technical Standard for Future Airborne Capability Environment (FACE), Edition 3.0.* The Open Group. Retrieved from www.opengroup.org/library/c17c

FACE Consortium. (May 2020). *Reference Implementation Guide for FACE™ Technical Standard, Edition 3.0, Volume 2: Computing Environment.* The Open Group.

PEO Aviation. (2018). Rapid Integration Framework (RIF) Demonstration Information Packet. *Proceedings of the 2018 September US Army FACE™ Technical Interchange Meeting.* Huntsville, AL: The Open Group.

Price, S. P., & Edwards, C. J. (2017). *A Common Command Interface for Interactive FACE Units of Conformance.* The Open Group.

__Multiple Transport Implementations__

About the Author(s)

Christopher J. Edwards has been working in the avionics industry for over 25 years, primarily on cockpit systems for military aircraft. In those years, he has served in leadership roles in System Architecture, Software Development, Requirements Capture, PVI development, Qualification Testing, and Project Management. Mr. Edwards work within the FACE Consortium has been as a principal author on both the FACE Conformance Policy and the FACE Technical Standard as well as many other consortium documents. Mr. Edwards currently leads the FACE Conformance Overview presentations and serves as a co-lead of the FACE Technical Working Group (TWG) Conformance Verification Subcommittee and as the facilitator of the FACE Verification Authority Community of Practice. Mr. Edwards serves as a MOSA Subject Matter Expert and is the Chief Architect and Systems Engineer for the Fixed Wing Family of Systems as well as other RIF related projects.

Steven P. Price has been working in avionics and embedded software for more than 30 years. He has worked on several different graphic user interfaces, including cockpit systems. He has been a leader in the design and implementation of some of these systems, along with being involved with the testing of some of these systems. Currently, Mr. Price is one of the Principal Software Engineers for RIF and the principal developer of the CMS Menu System. He is a co-lead on the FACE Transport sub-committee and FACE Verification Authority Subject Matter Expert (SME), along with involvement in other FACE sub-committees.

Rachel D. Moudy has been working in the missile defense and avionics industry for the past 9 years designing, developing, and integrating military software solutions. Currently, Mrs. Moudy supports the CMS team as the Software Engineering Lead and the Systems MBSE Lead for the Fixed Wing Family of Systems. She continues to acquire knowledge of user and design interactions to improve and create innovative solutions. Mrs. Moudy is currently pursuing a Master of Science in Human Factors with a concentration in Aerospace to ensure warfighter's behavior is captured throughout designs.

Shaun Foley is a senior software engineer at Skayl. He has 15 years of experience as a distributed systems consultant for defense and commercial customers in North America and Europe. He appreciates the integration and interoperability challenges at all levels of abstraction: the need for consistent data definitions, the need to maintain legacy functionality, and the pragmatic constraints of real-time and embedded platforms.

About The Open Group FACE™ Consortium

The Open Group Future Airborne Capability Environment™ Consortium (the FACE™ Consortium), was formed as a government and industry partnership to define an open avionics environment for all military airborne platform types. Today, it is an aviation-focused professional group made up of industry suppliers, customers, academia, and users. The FACE Consortium provides a vendor-neutral forum for industry and government to work together to develop and consolidate the open standards, best practices, guidance documents, and business strategy necessary for acquisition of affordable software systems that promote innovation and rapid integration of portable capabilities across global defense programs.

Further information on the FACE Consortium is available at www.opengroup.org/face.

About The Open Group

The Open Group is a global consortium that enables the achievement of business objectives through technology standards. Our diverse membership of more than 800 organizations includes customers, systems and solutions suppliers, tools vendors, system integrators, academics, and consultants across multiple industries.

The mission of The Open Group is to drive the creation of Boundaryless Information Flow™ achieved by:

- Working with customers to capture, understand, and address current and emerging requirements, establish policies, and share best practices
- Working with suppliers, consortia, and standards bodies to develop consensus and facilitate interoperability, to evolve and integrate specifications and open source technologies
- Offering a comprehensive set of services to enhance the operational efficiency of consortia
- Developing and operating the industry's premier certification service and encouraging procurement of certified products

Further information on The Open Group is available at www.opengroup.org.