Advanced Scientific Computing Research

Computer Science

TASCS Final Report

Svetlana Shasharina* [*]

Tech-X Corporation

11/30/2010

[*] *720-563-0322, sveta@txcorp.com*

1. Introduction

Tech-X was the only industry partner of The Center for Technology for Advanced Scientific Component Software (http://tascs-scidac.org/). The main technology of TASCS in the Common Component Architecture or CCA (www.cca-forum.org*)*. This project started in 11/15/06 with the total Tech-X funding $346,386. The main thrusts for Tech-X were "outreach" and "usability."

TASCS was terminated after the mid-term review but the university and industry partners were given till November 2010. This the summary of Tech-X work throughout the project.

2. Travel

Tech-X participated in all CCA forum meetings and presented our work at CCA forum presentations.

3. Outreach to Physics Applications

Accelerator Physics Applications (COMPASS)

The Synergia2 Advanced Accelerator Simulation projects (http://cd-amr.fnal.gov/aas/Advanced_Accelerator_Simulation.html) which is a part of the COMPASS SciDAC project (https://compass.fnal.gov/) uses TxPhysics (http://www.txcorp.com/products/TxPhysics/index.php) which is cross-platform library of computational modules for modeling how charged particles interact with their environment which facilitated coupling Synergia with TxPhysics. Tech-X Corporation has looked into the applicability of using the CCA tool OnRamp (https://outreach.scidac.gov/projects/cca-onramp/) to generate TxPhysics interfaces for calling TxPhysics from Synergia2. As a result of the OnRamp/TxPhysics work, the process of generating SIDL interfaces proved to be much easier using On Ramp compared to manual code generation.

Fusion Applications (FACETS)

The fusion application that we concentrated on was Framework Application for Core-Edge Transport Simulations (FACETS, http://www.scidac.gov/fusion/fullscale.html) the SciDAC project lead by Tech-X. The primary focus was to use Babel, the CCA language interoperability tool to communicate between the C++ FACETS framework and physics codes written in other programming languages, such as Fortran (77,90, 95), C and Python.

The FACETS components interfaces are defined using SIDL (Scientific Interface Definition Language from Babel). For example, here is the general initialize/finalize interface:

```
package facets version 0.1 {
        interface FacetsIfc {
                int readParams(in string inFile);
                int setMpiComm(in long comm);
                int setLogFile(in string logFile);
```

[*] *720-563-0322, sveta@txcorp.com*

```
                int getRankOfInterface(in string ifcName, out int rank);
                int initialize();
                int update (in double tnew);
                int dumpToFile(in string fname);
                int restoreFromFile(in string fname);
        }
}
```

Here is the 1D data access interface (there is also 2D and ND):

```
package facets version 0.1 {
        interface Facets0dIfc {
                int get0dDouble (in string varname, out double value);
                int set0dDouble (in string varname, in double value);
                int get0dDoubleAtLocation (in string varname, in
rarray<double> location(dim), in int dim, out double value);
                int set0dDoubleAtLocation (in string varname, in
rarray<double> location(dim), in int dim, in double value);
                int get0dInt (in string varname, out int value);
                int set0dInt (in string varname, in int value);
                int get0dIntAtLocation (in string varname, in rarray<double>
location (dim), in int dim, out int value);
                int set0dIntAtLocation (in string varname, in rarray<double>
location (dim), in int dim, in int value);
                int get0dString (in string varname, out string value);
                int set0dString (in string varname, in string value);
        }
}
```

The codes that were brought up into FACETS include FORTRAN transport modules (https://ice.txcorp.com/trac/fmcfm), fluxgrid (FORTRAN), and UEDGE (Python). The biggest achievement of Tech-X here was implementing struct support in Babel which is now in the Babel code base. Some details about this work are given in the next subsection.

Babel Struct Support (for FMCFM)

The current method of C++ to Fortran interoperability relies on the FC_FUNC autotools macro for name-mangling and uses a contiguous piece of memory on the C++ side that is manually aligned with the Fortran *derived types* for language interoperability. A key benefit to using Babel is that once a SIDL definition is in place there then exist common interfaces so that different models may be``plugged-in" to FACETS. A domain scientist can easily prototype a model (for example in Python) and test it using the common interface definition without implementing her own glue and integration code.

The transport models contain derived types with primitive types and statically allocated array members and we use SIDL to provide exact mappings of types (Figure 1). The one-to-one correspondence (for example statically allocated Fortran arrays map to SIDL r-arrays) retains computational efficiency and facilitates a straightforward conversion to SIDL and Babel. Figure 2 shows parts of the SIDL file for the transport models and how *structs* and *classes* are organized.

SIDL Type	F2003 type(kind)
BOOLEAN	logical(c_bool)
CHAR	character(c_char)
INT	integer(c_int32_t)
LONG	integer(c_int64_t)
FLOAT	real(c_float)
DOUBLE	real(c_double)
FCOMPLEX	complex(c_float_complex)
DCOMPLEX	complex(c_double_complex)
OPAQUE	integer(c_int64_t)
STRING	type(c_ptr)
ENUM	integer(c_int64_t)

Figure 1. SIDL primitive types mapped to their corresponding F2003 iso_c_binding type and kind.

In using SIDL to define common interfaces we are able to impose an object oriented structure for better code organization and ease of implementation. SIDL supports multiple inheritance of interfaces and single inheritance of implementation. We define a base class, which contains common implementation to all of the Transport Module Fortran codes that are being integrated (input, output) while the interfaces define the minimum types that must be used in any particular implementation. See the portion of SIDL code that defines a base class and interfaces and includes an implementation for the GLF23 module (Figure 3).

```
package fmcfmWrap version 1.0{
  struct MagGeom{};
  struct SurfVars{};
  struct Gradients{};
  struct Flags{
       ...
       double              glfCnu;
       rarray<int,1>       glfIflagin(5);
       rarray<double,1>    glfXparam(30);
       int                 mmmNroot;
       ...
};
  struct diflux{};
```

Figure 2. Extracts from the SIDL file used to wrap the transport models for the FACETS integration.

Initial prototype and full implementations that couple the FACETS C++ code with legacy Fortran code indicate that Babel provides a more general solution to the coupling problem than what is currently in place. Babel not only provides multi-platform and multi-compiler support, but we are finding that there are fewer levels of subroutine calls that are exposed to the user when using Babel. In effect the``interoperability glue" code required to combine languages is hidden better than it was in the original implementation. Other benefits include that 1) the cross-language boundary is more obvious to the user and 2) using Babel makes it easier for a user to focus on implementing algorithms in various languages without having to worry about the mechanisms associated with language interoperability and 3) when a legacy code has been “babelized" with SIDL, it can be easily reused by other codes that require the same module.

```
package fmcfmWrap version 1.0{

  class fmcfmIO{
    void  openFile(  in        string
infile,
                  in        str ing
status,
                  out           int
funit);
    void     closeFile(inout     int
funit);
  }

  interface fmcfmRun{
    void initFlags();
    void setFlags();
    void         calcFlux(         in
typeMod.magGeom          eqMg,
                  in
typeMod.allSpecies       species,
                  inout
typeMod.transSpecies     outFlux,
                  out           int
ierr);
    void dumpFlags(out   int
                outUnit);
  }

  class  fmcfmGlf23  extends  fmcfmIO
implements-all fmcfmRun{

    void initFlags[glf23]       (out
glf23Mod.glf23Flags      flags);
    void initFlags[glf23Extra] (out
glf23Mod.glf23FlagsExtra flags);

    void        setFlags[glf23](out
glf23Mod.glf23Flags      flags,
                         out
glf23Mod.glf23FlagsExtra
extraFlags);

    void         calcFlux[glf23](in
typeMod.magGeom          eqMg,
                         in
typeMod.allSpecies       species,
                         inout
typeMod.transSpecies     outFlux,
                         out     int
ierr,
                         inout
anomTypeMod.anomSurfVars sV,
                         in
glf23Mod.glf23Flags      flags);
  }
}
```

Figure 3. SIDL class and interface definitions and an example of the GLF23 transport module implementation.

Babel for FLUXGRID

We have also created SIDL interfaces for *fluxgrid* – a F90 code that generates magnetic flux-aligned grids and associated geometric quantities. *Fluxgrid* is currently used as part of the workflow steps in FACETS. By having both python and C++ wrappings, we now have both an improved workflow (with python driven fluxgrid and plotting capabilities) and tight-coupling capabilities. Both capabilities are useful depending on the case run, and BABEL-generated bindings provides flexibility to the FACETS project.

Babel for UEDGE

UEDGE is a physics component in FACETS that represents the edge of the tokamak plasma. Python is used in UEDGE and we use Babel to communicate between C++ and Python. The server Python bindings come from Facets0dIfc.sidl and FacetsIfc?.sidl files shown above. Since Babel generates server skeletons and the implementation is supposed to be inserted in the generated code, we currently do the following.

Upon any change in sidl interfaces, we generate Python bindings. Next we edit and add implementation and disable bindings regeneration using particular Babel flags. For example, for we had to go through the following (as we modified interfaces):

1. Define uedge to implement the desired interfaces:

```
package ue version 0.2 {

        class Uedge implements-all facets.FacetsIfc, facets.Facets0dIfc { }

}
```

2. Generate Python server stubs:

```
  <babel_location>/bin/babel --server=python uedge.sidl
```

This will create Uedge_Impl.py.

3. Modify impl file: get rid of _getStub method and make the impl to inherit from ue.Uedge_Aux.Uedge and modify _init so that the class looks like:

```
  class Uedge(ue.Uedge_Aux.Uedge):    def __init__(self, IORself = None):
ue.Uedge_Aux.Uedge.__init__(self, IORself)     # other methods
```

4. Implement Python methods. The key here is to remember that Python does not allow out variables, so the out variable should be packages in the return tuple consisting of the original sidl return and all out variables. For example, get0dDouble should look like:

```
  def get0dDouble(self, varname):

        value = (0, self.uedge.getDouble(varname))

        return value
```

4. Outreach outside of CCA

The main philosophy of CCA is the use of components for “plug-and-play” computing and providing language interoperability tools. When TASCS was terminated, Tech-X still had sixmonths of extra time to address the possible limitations of this project. We contacted TASCS PI David Bernholdt (ORNL) for directions, and were approved to try new things.

First of all we decided to explore the use of components for high-performance I/O (use of ADIOS for FACETS) and second, we looked at the middleware (Data Distribution Service, or DDS, http://www.omgwiki.org/dds/) that came out of the OMG community and became a successor of CORBA. Since CCA heavily borrowed CORBA ideas, it would make sense to look at DDS and see how it can be used in scientific applications. That is why we explored how would one generate Python bindings for DDS and implementing security within DDS systems. The details follow.

Componentized I/O (ADIOS) for use in FACETS

The ADaptable IO System (ADIOS, http://www.olcf.ornl.gov/center-projects/adios/) is a set of data abstractions and APIs that provide flexible methods for encapsulating and transporting scientific simulation data. ADIOS, developed jointly by ORNL and Georgia Institute of Technology's Center for Experimental Research in Computer Systems, is designed with very large high-performance computing (HPC) architectures in mind. It is portable and scalable, and provides a number of different methods for transporting data from the simulation to the abstracted I/O layer. Transport methods can be specified in an external file, so the I/O behavior of simulation codes can be rapidly changed from run to run without time-consuming recompilation. Different transport methods can be specified for different objects in a single simulation, adding more flexibility to how data is handled.

Our goal was to evaluate ADIOS as a possible I/O component for physics applications because of its flexibility, and large number of transport methods that are supported, including asynchronous methods that use RDMA to directly access simulation memory. We recognize that asynchronous methods will be required in the future in order to do in situ visualization and to minimize dump times for very large simulations (100k cores).

The main goal of this effort was to identify if the ADIOS I/O components are suitable for implementing I/O of such physics applications as VORPAL (http://www.txcorp.com/products/VORPAL/) heavily used in COMPASS SciDAC) and FACETS. These codes use HDF5 as the I/O format and in addition impose a set of standards on the HDF5 metadata that allows users and tools to identify and interpret data of interest. For example, the Visualization Schema (https://ice.txcorp.com/trac/vizschema/wiki/WikiStart) provides metadata that describes how to visualize simulation data arranged in groups of attributes, in the HDF5 sense.

We identified a number of features that will need to be added to ADIOS in order for use to be able to produce output HDF5 files using particular markup.

- Some of these attributes, such as spatial dimensions of the simulation, are stored as arrays. This is encouraged in HDF5, but is not compatible with the current structures of ADIOS data files.
- In addition, scalar an string attributes that are determined at run time are currently required by the ADIOS API to correspond to a dataset, and there is no way to just write an attribute without it referencing an existing dataset. This has to do with the way that ADIOS structures its data format, so that data access is very efficient with large, distributed datasets. However, in order to utilize ADIOS for our purposes, we will need to add support for more generic writing of I/O objects.

- In addition, we foresee the need to change the build system to cmake from autotoools in order to be able to build more flexibly on different operating systems (including windows), and the need to separate out MPI dependencies so that purely serial codes can be supported using ADIOS.

The conclusion of our evaluation is that ADIOS could be very useful, but needs to work with particular physics applications to ensure that their needs (like schemas) are supported. We also decided that Tech-X needs to define its I/O needs moving forward and to evaluate ADIOS and possibly other tools in that context. Supported output formats (and performance of required formats), data rearrangement during I/O and utility for coupling components and for sequential processing are some needs which come to mind along with obvious factors such as portability and scaling.

Python Bindings for DDS

Introduction

The PyDDS project was proposed to research and design Python mappings for DDS, which would allow Python applications to participate in DDS data exchanges easily. There is no standardized DDS Python mapping available and for developers to use DDS in their Python applications, they must first generate the topic-specific supporting code in another language such as C that can easily be bridged into Python. Software wrapper utilities such as SWIG then enable python applications to interact with these interfaces. Although this approach works fine for simple applications, it is hard to evolve and maintain, and is very application specific. The wrapper approach also requires the developers to be savvy at multiple technologies at the same time, which makes it harder to use.

Our effort is to focus on eliminating the need to generate application-specific wrapper for Python applications. The prototype will investigate how DDS topic-specific operations are mapped given a topic structure and name.

Design

We have used OpenSplice Community Edition version 5.3 (http://www.opensplice.com/) as the base DDS implementation for this project. The design involves identifying the core C++ communication layer of OpenSplice and creating a layer of Python bindings on top of it. This layer can then be used by Python applications directly for DDS communications and data transfer, which is much easier and straightforward when compared to generating the DDS bindings in C++ and then creating bindings to be used by Python applications. This approach also makes it possible for DDS applications (Publishers/Subscribers) written in other OpenSplice compliant languages to communicate seamlessly.

The design (Figure 4) is geared in a way that the Python application calls to OpenSplice are very similar to that of a C++ DDS application. As an example, the following code snippet shows how to create a DDS Domain Participant object from a Python DDS application:

```
import PyDDS


# Create DomainParticipant
dpf = PyDDS.DomainParticipantFactory.get_instance ()
```

Once the PyDDS library is implemented, it will be possible to implement an IDL compiler for generating Topic specific Python mappings which will use the PyDDS bindings underneath. The mappings will essentially derive from classed defined in PyDDS library to create type specific calls, data marshalling and demarshalling. The diagram on Figure 5 shows the flow of control to create a topic.

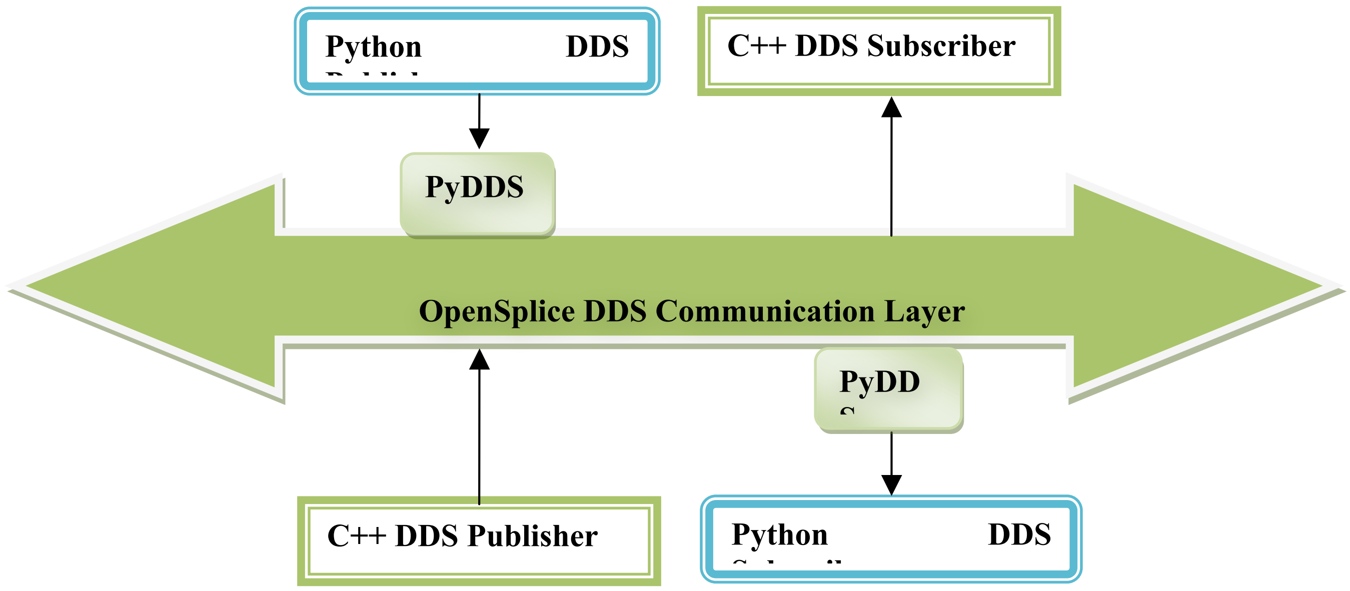

Figure 4. PyDDS Design.

OpenSplice's core communication classes

The primary task of this project involves determining the C++ classes from OpenSplice on top of which the Python bindings need to be created. The main classes of interest are:

- DomainParticipantFactory, DomainParticipant, DomainParticipantQos
- DataReader, DataReaderQos
- DataWriter, DataWriterQqs
- Publisher, PublisherQos
- Subscriber, SubscriberQos
- Topic, TopicQos, TypeSupport
- Sequence, InfoSeq

Almost all the classes mentioned above are derived from other classes, have multiple layers of inheritance and multiple inheritances. All the inheritance levels need to be taken care of in the bindings, and including all the classes, there are about 150 classes which need bindings to be created. As a specific example, the DomainParticipant class is a singleton, and its constructors and destructor are private. The DomainParticipantFactory class has a static method that needs to be exposed to get a handle to the DomainParticipant's singleton instance. All these complexities need to be taken care of while creating the Python bindings. We have explored using SWIG and Boost Python to create the bindings for the aforementioned classes.

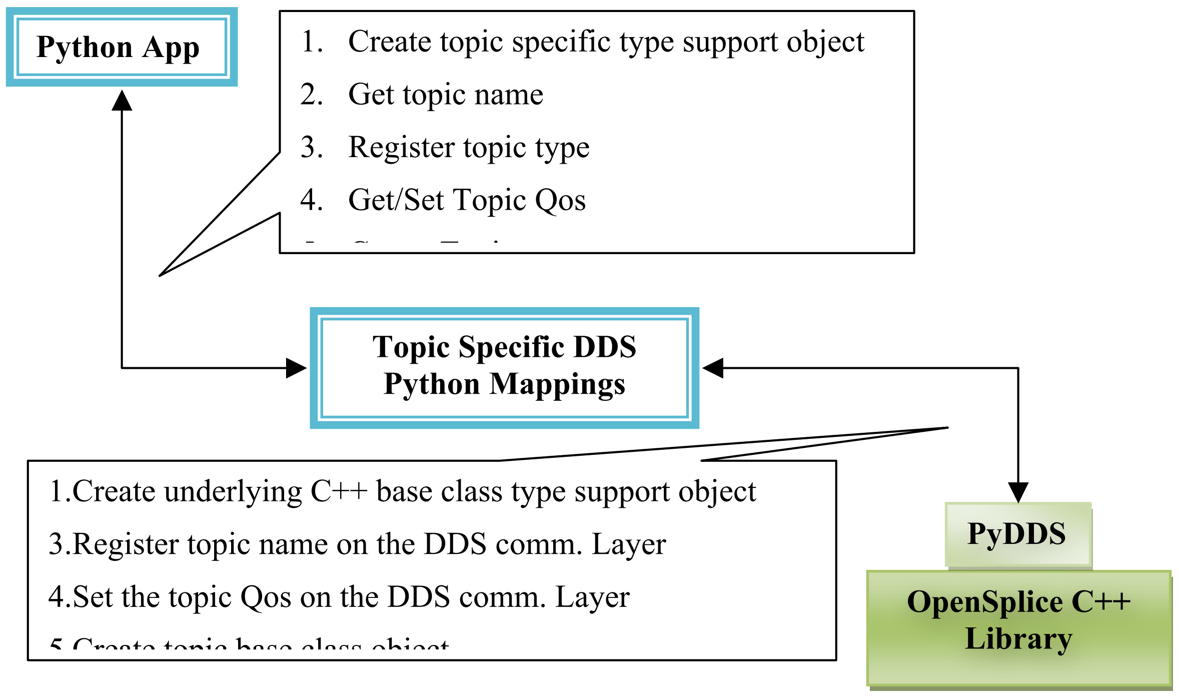
Python App
1. Create topic specific type support object
2. Get topic name
3. Register topic type
4. Get/Set Topic Qos
Topic Specific DDS Python Mappings
1.Create underlying C++ base class type support object
3.Register topic name on the DDS comm. Layer
4.Set the topic Qos on the DDS comm. Layer
PyDDS
OpenSplice C++ Library

Figure 5: Flow of control to create a topic from Python application using PyDD

SWIG

SWIG is a language interoperability software tool that connects programs written in C and C++ with a variety of high-level programming and common scripting languages. We explored using SWIG to generate bindings between the C++ core communication code of OpenSplice and Python. This approach seemed simpler at first, as it needed just a few header files to be included in the SWIG interface file, and passed on to SWIG to generate the bindings automatically. Since most of the classes are derived from other base classes, even the base class headers need to be processed, and for a well-formed complicated library like OpenSplice, it is essentially necessary to add almost all the header files to the SWIG interface file. SWIG does have an option of following all the includes, which makes the interface file simpler, but it goes down to the system file level, and has problems with system level macros that define the endian-ness, 32/64 bit machines, etc. To take care of these problems, manual definitions of these system information is needed in the SWIG interface file which is now machine specific.

Boost Python

Boost Python is a C++ library which enables seamless interoperability between C++ and the Python programming language. The bindings are coded in a C++ file which can then be compiled and linked to the underlying C++ code for which the bindings are created. The classes and methods need to be listed manually, and this can get a bit laborious, but also has a flexibility of exposing only the methods that are needed. In case of multiple levels of inheritance, all the levels must be exposed in the bindings, and special care should be taken in cases of Singleton classes, where the constructor and destructors are private.

Below is an example of how the Boost Python bindings will look for a single level of inheritance with Singleton classes, which have a static method to get an instance of the derived object as a base class pointer:

```
//
// Base class
//
#ifndef _BASE_
#define _BASE_
#include "Derived.h"
class Base {
  friend class Derived;
 public:
  virtual void a() = 0;

 private:  // Private constructor and destructor
  Base(){}
  ~Base();
};
#endif /* _BASE_ */
//
// Derived class
//
#ifndef _DERIVED_
#define _DERIVED_
#include "Base.h"
class Derived : public Base {
 public:
  virtual void a();
  static Base* get_instance();
 private:     // Private constructor and destructor
  Derived(){}
  ~Derived();
};
void Derived::a() {
}
Base* Derived::get_instance() {
  Derived* d = new Derived();
  Base* b = dynamic_cast<Base*>(d);
  return b;
}
#endif /* _DERIVED_ */
//
// Boost Python Bindings
//
#include "boost/python.hpp"
#include "/home/research/roopa/projects/pydds/examples/boost/Derived.h"
namespace bp = boost::python;
struct Base_wrapper : Base, bp::wrapper< Base > {
    virtual void a(  ){
        bp::override func_a = this->get_override( "a" );
        func_a(  );
    }
};
struct Derived_wrapper : Derived, bp::wrapper< Derived > {
    virtual void a(  ) {
        if( bp::override func_a = this->get_override( "a" ) )
            func_a(  );
        else
            this->Derived::a(  );
    }
    void default_a(  ) {
        Derived::a( );
    }
};
```

```
BOOST_PYTHON_MODULE(test){
    bp::class_< Base_wrapper, boost::noncopyable >( "Base", bp::no_init )
        .def(
            "a"
            , bp::pure_virtual( (void ( ::Base::* )(  ) )(&::Base::a) ) );
    bp::class_< Derived_wrapper, bp::bases< Base >, boost::noncopyable >( "Derived",
bp::no_init )
        .def(
            "a"
            , (void ( ::Derived::* )(  ) )(&::Derived::a)
            , (void ( Derived_wrapper::* )(  ) )(&Derived_wrapper::default_a) )
        .def(
            "get_instance"
            , (::Base * (*)(  ))( &::Derived::get_instance )
                /* undefined call policies */ )
        .staticmethod( "get_instance" );
}
```

The OpenSplice library uses C library underneath the C++ classes for DDS communications. The C code uses forward declarations of structures that are not available in the exposed header files. This needs further investigation, and is beyond the time scope of this project.

Future Work and Summary

We have modified the sources of the OpenSplice's community version to expose a few of the forward declarations in the Boost Python bindings. To finish up and to have a working library of Python bindings, we need to investigate more and discover ways to expose more of the underlying C code.

One option would be to come up with a simpler API on top of the communication layer and create the Python Bindings to only this simpler API which will then use OpenSplice underneath. With this option, there will be no need to modify any of the OpenSplice’s source code, and the PyDDS library can be used by both the community and the commercial versions of OpenSplice as long as the communication API.

Once this is done, we can implement either an IDL compiler for generating Python mapping, or a dynamic loader that better takes advantage of Python dynamic language nature. The loader approach will allow Python programs to “load up” the IDL definitions directly and generate the appropriate type-specific objects. Once the bindings are successfully created, and PyDDS library is built, it can be used successfully in various Python DDS applications seamlessly.

This work will then result in publish-subscribe software available to the scientific community using Python and could be used for implementing distributed loosely-coupled components systems. This would be a great substitute for the CCA technology for the case when scalability and RT issues are important.

Security in DDS

Introduction

All distributed technologies, like CCA, Web Services and CCA, face the same problem: they do not come with the prepackaged security implementation. Security (authentication, authorization and encryption) will be needed for all real component-based distributed systems. Since we found DDS very promising for such systems, we decided to investigate how would one implement security within DDS. Hence the goal of this effort was to test the feasibility of providing a secure transport service for the OpenSplice community DDS software.

We use two major open source components, version 5.3 of the OpenSplice community edition and version 1.0.0a of the OpenSSL toolkit. OpenSplice community edition is a well-regarded open source implementation of the Object Management Group's Data Distribution Service standard. OpenSSL is a widely used toolkit for building secure networking solutions. It is most widely known as the toolkit behind the secure connection (HTTPS) of web browsers and most web servers.

The basic design concept is to intercept network traffic to and from an OpenSplice DDS application and the computer's network connection. DDS messages destined for another computer are to be encrypted and signed before being sent out via the computer's network connection while messages coming into the computer will be verified and decrypted before being sent along to the DDS application.

Design

At the coarsest level, the architecture of this project consists of connection management code which serves as an interface between a running OpenSplice DDS application and the OpenSSL toolkit. All network traffic from OpenSplice is processed by the interface and passed to OpenSSL before it reaches the network. Incoming traffic traces a reverse path. Because OpenSSL is a toolkit, no modifications are necessary to the codebase. However, the OpenSplice system must be modified in order to redirect its network traffic.

OpenSplice Modification

The OpenSplice community edition funnels all of its network traffic through a small number of routines closely modeled on the unix socket networking paradigm (see Figure 6). These routines are compiled into a shared library object file. SecurDDS can take advantage of this architecture by replacing that shared library with one implementing the same interfaces but which communicates with a secure network layer instead of directly with the network (see Figure 7). This is the approach taken during this investigatory project.

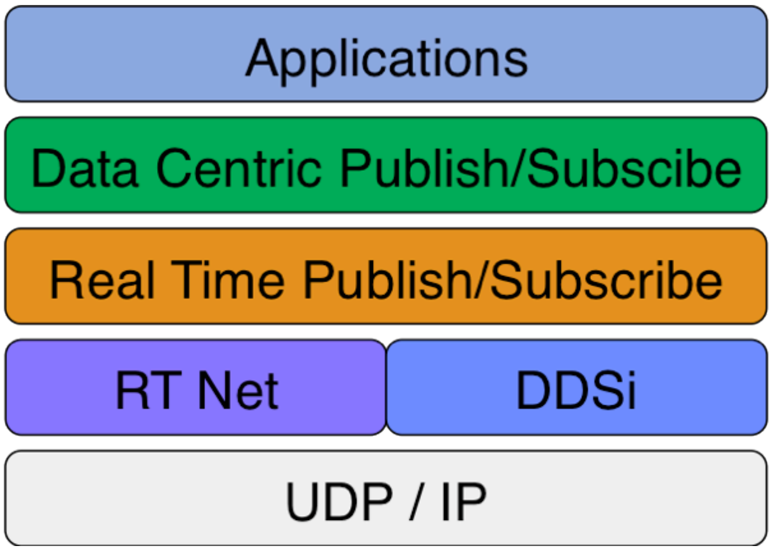

Figure 6. Overview of OpenSplice DDS Community Edition Architecture

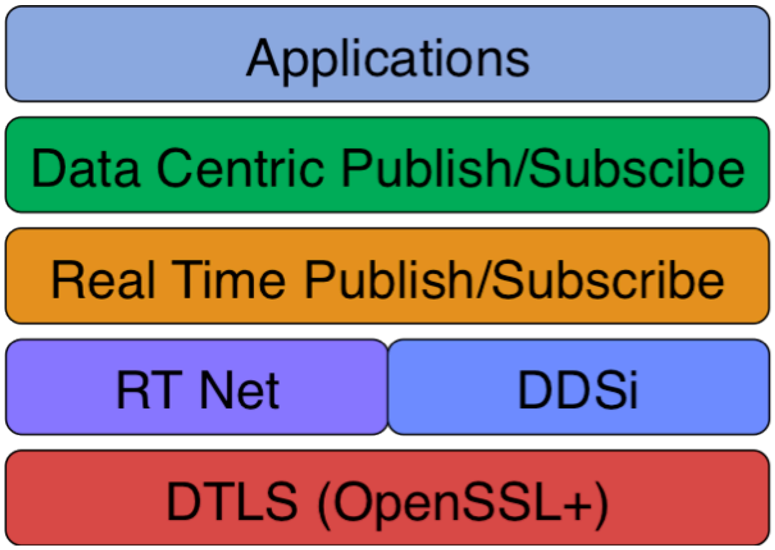

Figure 7: OpenSplice DDS with the lowest network layer replaced with a secure networking module

The Security Interface Module (SIM)

The job of the Security Interface Module is to manage secure connections on behalf of the DDS application. A secure connection contains a lot of state, usually referred to as a Security Association (SA). The SIM has to keep track of this state in order to correctly handle traffic to and from different connections. The tasks of the SIM are listed below.

- **Create outgoing connections:** When the DDS application creates an outgoing connection (usually by sending data on a newly-created socket), the SIM initiates a security negotiation with the indicated destination and, upon successful completion, sends the data traffic.

- **Handle outgoing traffic:** When traffic is received from the DDS application, the SIM first checks to see if there is a secure connection for that traffic. If so, the SIM sends the traffic to the correct SA for encryption and signing before it is sent out.
- **Listen for new incoming connections:** When the DDS application indicates that it is going to listen on a port (by binding to a socket), the SIM listens for an incoming connection and when one arrives, conducts a security negotiation. Secured traffic is then passed to the DDS application. The SIM configures OpenSSL to recognize future incoming traffic on this newly created SA.
- **Handle incoming traffic:** When encrypted traffic is received, it is handled by OpenSSL functions which check the signature and decrypt the data payload. If everything is OK, it passes the data to the SIM which passes it to the DDS application

Security design

For a production system, security parameters will depend on topics, publishers, and subscribers. However, for this prototype, we the same security parameters for all connections. In order to create a secure communication channel between two entities, several negotiation steps must be completed:

1. Mutual authentication: Each party to a security negotiation must be able to positively identity that it is communicating with a known peer. For this experiment, certificates were created to mimic commonly used authentication protocols (e.g., DOE Grids Certificates).

2. Negotiation of security parameters: The parties negotiating a secure connection must agree on encryption and signing algorithm as well as the strength (i.e., key length) to use.

3. Exchange of key material: Before encryption and digital data signing can begin, the peers have to exchange signed sets of random numbers used to securely create encryption and authentication keys.

The SIM manages this process by configuring an OpenSSL connection before negotiation begins. This process happens independently on both sides of a future secure connection where one side is performing task 1 and the other is performing task 3. The actual negotiation is then run by OpenSSL functions.

Future work and Conclusions

We concluded that it is feasible to implement security within DDS. Possible directions for future research could include the following tasks:

- Determine the best way to configure DDS applications to use our secure version of OpenSplice DDS to enforce local security policies. Ideally, we would like to implement a configuration scheme identical to or very close to that used by the OpenSplice enterprise edition.
- Investigate whether the current method of placing the SIM at the lowest network layer of the OpenSplice stack is the best for long-term security code development. PrismTech has implemented security in its enterprise edition by creating a replacement for the real-time networking layer ('RT Net' box in Figure 6). While this would entail more work at the front end (the real-time networking layer is more complex than the UDP/IP layer), it would provide direct access to the DDS-specific data attributes which would ease fine-

grained control over security protocol decisions. The resulting architecture would resemble Figure.

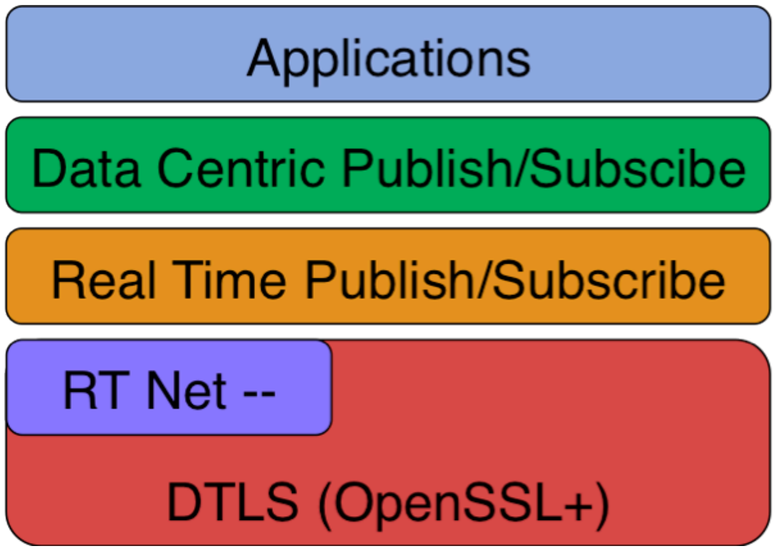

Figure 8. Secure verion of OpenSplice where the interface is at the real time networking layer of the OpenSplice architecture.

5. Usability of CCA tools

Tech-X has also worked to promote further use of CCA tools by providing education to non-computer scientists regarding component technologies. We have worked closely with domain scientists to integrate CCA tools with existing build systems, we have ported Babel to x86_64, provided bug fixes and continue to provide internal support to Tech-X. We have also created a special svn repository with canonical examples of how to work with CCA components in general, using Babel and how to use Bocca.

Introducing Babel into existing build system adopted by FACETS brings complications, as it requires checking in generated files (babel.make) into repositories. To address this, we have developed a workflow and corresponding file structure that allows us to side-step the inclusion of babel.make in repositories, requires minimal modification to our existing build systems and retains the autotools build steps (configure, make and make install). Adding this functionality to our build systems increases the software usability by potential clients, hides the inclusion of Babel as an additional software dependence and reduces software version updates and checks.

6. Publications

Nanbor Wang, Rooparani Pundaleeka and Johan Carlsson, "Distributed Components for Integrating Large-Scale HPC Applications", in Proceedings of 2007 Workshop on HPC Grid Programming Environments and Components and Component and Framework Technology in High-Performance and Scientific Computing (HPC-GECO/Compframe 2007), Montreal, Quebec, Canada, October, 2007.

S. Shasharina, N.Wang, S. Muszala, R. Pundaleeka, "Grid and Component Technologies in Physics Applications," proceedings of ICALEPCS 2007, October 15-19, 2007, Knoxville, TN.

Nanbor Wang, Paul Hamill, Fang Liu, Steve Tramer, Roopa Pundaleeka, and Randall Bramley, "Integrating Large-scale Distributed and Parallel HPC (DPHPC) Applications using a Component-Based Architecture", in the Proceedings of 2008 Workshop on Component-Based High-Performance Computing (CBHPC 2008, Karlsruhe, Germany, October, 2008.

S. Muszala, T. Epperly, and N. Wang, "Babel Fortran 2003 Binding for structured data types.", Springer Lecture Notes in Computer Science. Paper accepted, pending publication. PARA 2008-9th International Workshop on State-of-the-Art in Scientific and Parallel Computing. May, 2008. Trondheim, Norway.

Svetlana Shasharina, John R. Cary, Ammar Hakim, Gregory R. Werner, Scott Kruger, Alex Pletzer, "FACETS – A Physics Driven Parallel Component Framework", in the Proceedings of 2008 Workshop on Component-Based High-Performance Computing (CBHPC 2008, Karlsruhe, Germany, October, 2008.

Fang Liu, Nanbor Wang, Roopa Pundaleeka, Randall Bramley; "Enabling Concurrent Data Processing for Fusion Simulation Through Distributed CCA Components", ISCA 22nd International Conference on Parallel and Distributed Computing and Communication Systems, PDCCS-2009, September 24-26, 2009, Louisville, Kentucky, USA.

S. Muszala, J. Amundson, L. McInnes, B. Norris, "Two-tiered component design and performance analysis of Synergia2 accelerator simulations", 2009 Workshop on Component-Based High Performance Computing (CBHPC 2009).

7. Summary

During this project, Tech-X team's goal was to connect the Common Component Architecture tools to the scientific applications which we are involved in: accelerator physics and fusion. We found that the most useful tool from CCA was the Babel language interoperability tool. We ended up using Babel to specify FACETS interfaces and implement interaction between the FACETS C++ framework and multiple physics components.

Once the project was canceled, we extended our research to evaluate other than CCA technologies, which were close to CCA "in spirit": high-performance I/O components and Data Distribution Service. We found these tools very useful for the physics applications of interest and intend to pursue their use in our future projects.

8. People

There were many Tech-X people involved in this work: Svetlana Shasharina (Tech-X PI). Nanbor Wang, Johan Carlsson, Stefan Muszala, Rooparani Pundaleeka, Stephen Goldhaber, Seth Veitzer, Srinath Vadlamani, Alexander Pletzer and Scott Kruger.