QLDDS - Data Distribution Service for QuantLib

Oct 2013

Mike Kipnis

mike.kipnis @gmail.com

Summary

QLDDS is an open source project that simplifies the use of QuantLib in the distributed environment via OpenDDS(www.opendds.org). Using QLDDS, the functionality of the QuantLibAddin/C++ interface may be distributed across multiple computers running different operating systems, in real-time. This distribution is accomplished by publishing and subscribing to DDS data type messages that trigger corresponding QuantLibAddin calls with the received data. OpenDDS also allows to expose the functionality provided by QuantLibAddin to other languages including Java, C#, Python and Erlang.

Content

QLDDS package includes OMG IDL data types and corresponding DataReaders autogenerated from QuantLibAddin for C++. It includes examples and convenience classes that simplify interaction between OpenDDS, CORBA and QuantLibAddin.

Middleware

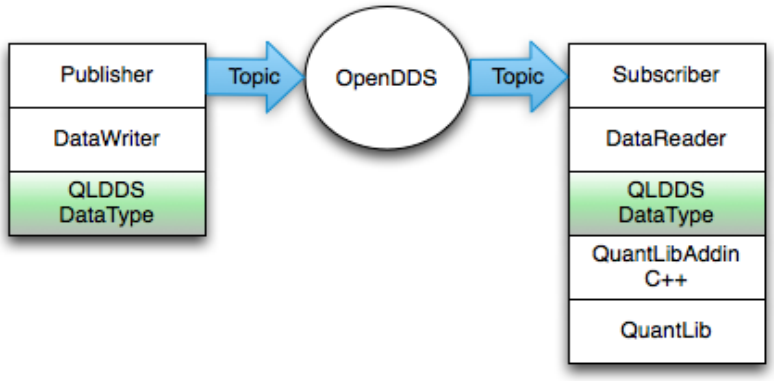
Data Distribution Service

QLDDS utilizes OpenDDS(www.opendds.org) as a data distribution middleware. OpenDDS is an open source implementation of the OMG Data Distribution Services for Real-Time System specification. Please review OpenDDS Developers Guide to get familiar with the basic DDS concepts before continuing with this document.
http://download.ociweb.com/OpenDDS/OpenDDS-latest.pdf

Package

Overview

QLDDS contains converted IDL data types and corresponding data readers from the QuantLibAddin/C++ function definitions. Data samples from converted IDL data types are published to subscribed data readers via DDS topics. Once data samples are received, they are converted into QuantLibAddin objects by invoking corresponding QuantLibAddin/C++ calls. IDL data types and data readers are auto-generated and are located in $QLDDS_ROOT/Addins/OpenDDS. *(See Fig 1.0)*



(fig 1.0)

Project Dependences Dependences

Project	**Version**
OpenDDS	3.4.1
QuantLib	1.2.1/1.3
GenSrc	1.2.0
ObjectHandler	1.2.0
QuantLibAddin	1.2.0
QuantLibAddinCpp	1.2.0

Unix:

All QuantLib dependences need to be configured, compiled and installed(make install). OpenDDS needs to be configured and compiled prior to building QLDDS.

Windows:

All QuantLib dependences have to be compiled and linked with dynamic run-time libraries support(**gd**).

IDL

Data Types

OpenDDS data types are defined using IDL(*see Defining the Data Types in OpenDDS Developer Guide*).
The auto generated IDL data types are located in $QLDDS_ROOT/Addins/OpenDDS.

Example of qlSwapRateHelper2 Data Type from ratehelpers.idl:

```
#pragma DCPS_DATA_TYPE "ratehelpers::qlSwapRateHelper2"
#pragma DCPS_DATA_KEY "ratehelpers::qlSwapRateHelper2 instanceID"

  struct qlSwapRateHelper2 {
        string instanceID;

          string ObjectId;
          double Rate;
          string Tenor;
          string Calendar;
          string FixedLegFrequency;
          string FixedLegConvention;
          string FixedLegDayCounter;
          string IborIndex;
          double Spread;
          string ForwardStart;
          string DiscountingCurve;
          boolean Permanent;
          boolean Trigger;
          boolean Overwrite;
  };
```

OpenDDS provides a compiler that generates the code to marshal and demarshal idl messages, as well as type support code for data readers and writers (see Processing the IDL in OpenDDS Developers Guide).

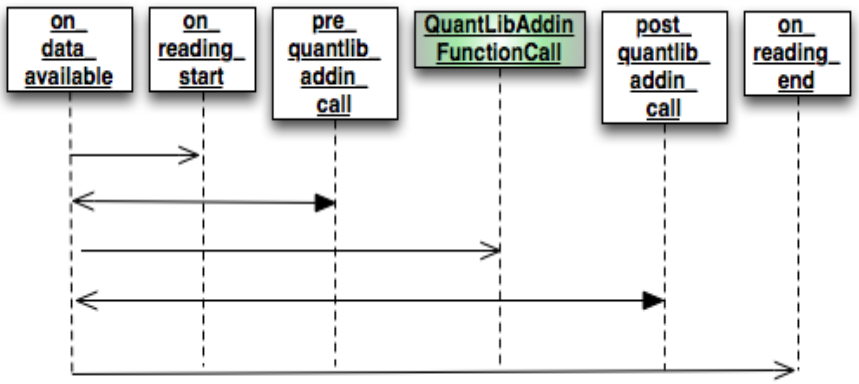
Example of using generated C++ code for qlSwapRateHelper2:

```
ratehelpers::qlSwapRateHelper2 swap_rate_helper2;

swap_rate_helper2.ObjectId = CORBA::string_dup(“IRS_10Y”);
swap_rate_helper2.Rate = 0.03165;
swap_rate_helper2.Tenor = CORBA::string_dup(“10Y”);
swap_rate_helper2.Calendar = CORBA::string_dup("TARGET");
swap_rate_helper2.FixedLegFrequency = CORBA::string_dup("Annual");
swap_rate_helper2.FixedLegConvention=CORBA::string_dup("Unadjusted"
 );
swap_rate_helper2.FixedLegDayCounter = CORBA::string_dup("30/360");
swap_rate_helper2.IborIndex =  CORBA::string_dup("Libor");
swap_rate_helper2.ForwardStart =  CORBA::string_dup("0D");
swap_rate_helper2.Spread = 0.0;
swap_rate_helper2.Overwrite = true;
```

DataReaders **Default Data Readers**

Default Data Readers are derived from qldds::DataReaderListener. qldds::DataReaderListener is an abstract class defined in DataReaderListener.hpp, located in $QLDDS_ROOT/qldds_utils. **qldds**::DataReaderListener is derived from **DDS**::DataReaderListener and has a number of additional pure virtual functions that will be called during data reading. Thes calls include reading start, reading end, pre and post QuantLibAddi function calls, as well as calls during exceptions. *(Fig 2.0)*



(Fig 2.0)

Default DataReaders are autogenerated from QuantLibAddin/C++ XML function definitions using qldds_gensrc.py. Default Data Readers are included in the package and are located at $QLDDS_ROOT/Addins/OpenDDS.

Below is an incomplete class declaration of a default data reader for qlSwapRateHelper2 data type. The complete class declaration can be found in $QLDDS_ROOT/Addins/OpenDDS/ratehelpersDataReaderListenerImpl.hpp

```
class qlSwapRateHelper2DataReaderListenerImpl
   : public virtual OpenDDS::DCPS::LocalObject
<qldds::DataReaderListener< ratehelpers::qlSwapRateHelper2, std::string
> >;
```

Note that qldds::DataReaderListener takes two template parameters. First parameter indicates QLDDS datatype which is ratehelpers::qlSwapRateHelper2 in case of

qlSwapRateHelper2DataReaderListenerImpl. Second parameter indicates the type returned by a particular QuantLibAddin call whic is a std::string returned by QuantLibAddinCpp::qlSwapRateHelper2 in this case. A reference to an instance of an object of the first template parameter is being passed to pre_quantlib_addin_call and post_quantlib_addin_call. A reference to an instance of an object of the second template parameter is being passed to post_quantlib_addin_call.

Below is a snapshot of an autogenerated code of on_data_available for qlSwapRateHelper2DataReaderListenerImpl. Complete declaration can be found in $QLDDS_ROOT/Addins/OpenDDS/ ratehelpersDataReaderListenerImpl.cpp

```
ratehelpers::qlSwapRateHelper2 obj;
DDS::SampleInfo si ;
DDS::ReturnCode_t status = obj_dr->take_next_sample(obj, si) ;

if ( status == DDS::RETCODE_OK && si.valid_data == true )
{
  if ( si.valid_data == true )
  {
    ++count;

    if ( pre_quantlib_addin_call( reader, si, obj ) )
    {
      std::string returnObject;;

      try {

        ACE_Guard<ACE_Mutex> guard( get_ACE_Mutex() );

        returnObject = QuantLibAddinCpp::qlSwapRateHelper2 (
        obj.ObjectId.in(),
        obj.Rate ,
        obj.Tenor.in(),
        obj.Calendar.in(),
        obj.FixedLegFrequency.in(),
        obj.FixedLegConvention.in(),
        obj.FixedLegDayCounter.in(),
        obj.IborIndex.in(),
        obj.Spread ,
        obj.ForwardStart.in(),
        obj.DiscountingCurve.in(),
        static_cast<bool>(obj.Permanent),
        obj.Trigger,
        static_cast<bool>(obj.Overwrite) );

      } catch ( std::exception& e )
      {
        on_std_exception( reader, obj, e );
        continue;
      }
```

```
        if ( !post_quantlib_addin_call( reader, obj, returnObject ) )
          break;
      }
    }
```

Sync

Threading

OpenDDS creates a separate thread to handle transport I/O events(*see Threading in OpenDDS Developers Guide*). Since QuantLib, ObjectHandler and QuantLibAddin for C++ are not thread safe libraries, all calls to these components must be synchronized when called from different threads. Constructors of Default Data Readers accept a reference to an ACE_Mutex object that will be locked before making a QuantLibAddin for C++ call and released after the call is complete.

Domain Participant

BasicDomainParticipant

BasicDomainParticipant is a class that provides convenience functions to create publishers, subscribers, thread safe default data readers and data writers. This class is essentially a wrapper around DDS API. BasicDomainParticipant can be found in $QLDDS_ROOT/qldds_utils/BasicDomainParticipant.hpp

Example of a publisher for ratehelpers::SwapRateHelper2 :

```
DDS::DomainParticipantFactory_var dpf =
       DDS::DomainParticipantFactory::_nil();
DDS::DomainParticipant_var participant =
       DDS::DomainParticipant::_nil();

dpf = TheParticipantFactoryWithArgs(argc, argv);

qldds_utils::BasicDomainParticipant participant( dpf, IRS_DOMAIN_ID );
participant.createPublisher();

DDS::Topic_var swap_rate_helper2_topic =
                  participant.createTopicAndRegisterType
                  < ratehelpers::qlSwapRateHelper2TypeSupport_var,
                   ratehelpers::qlSwapRateHelper2TypeSupportImpl >
                  (  SWAP_RATE_HELPER2_TOPIC_NAME );

ratehelpers::qlSwapRateHelper2DataWriter_var swap_rate_helper2_dw =
               participant.createDataWriter
               < ratehelpers::qlSwapRateHelper2DataWriter_var,
               ratehelpers::qlSwapRateHelper2DataWriter >
               ( swap_rate_helper2_topic );
```

```
ratehelpers::qlSwapRateHelper2 swap_rate_helper2;
DDS::InstanceHandle_t swap_rate_helper2_instance =
swap_rate_helper2_dw->register_instance( swap_rate_helper2 );

swap_rate_helper2.ObjectId = CORBA::string_dup("IRS_10Y");
swap_rate_helper2.Rate = 0.03165;
swap_rate_helper2.Tenor = CORBA::string_dup("10Y");
swap_rate_helper2.Calendar = CORBA::string_dup("TARGET");
swap_rate_helper2.FixedLegFrequency = CORBA::string_dup("Annual");
swap_rate_helper2.FixedLegConvention=CORBA::string_dup("Unadjusted"
 );
swap_rate_helper2.FixedLegDayCounter = CORBA::string_dup("30/360");
swap_rate_helper2.IborIndex =  CORBA::string_dup("Libor");
swap_rate_helper2.ForwardStart =  CORBA::string_dup("0D");
swap_rate_helper2.Spread = 0.0;
swap_rate_helper2.Overwrite = true;

int ret = swap_rate_helper2_dw->write( swap_rate_helper2,
swap_rate_helper2_instance );

 if (ret != DDS::RETCODE_OK) {
         ACE_ERROR ((LM_ERROR, ACE_TEXT("(%P|%t) ERROR: Swap write
returned %d.\n"), ret));
 }
```

Example of subscriber for ratehelpers::SwapRateHelper2 with a default DataReader:

```
DDS::DomainParticipantFactory_var dpf =
DDS::DomainParticipantFactory::_nil();
DDS::DomainParticipant_var participant =
DDS::DomainParticipant::_nil();

QuantLibAddinCpp::initializeAddin();
QuantLib::Calendar calendar = QuantLib::TARGET();
QuantLib::Date settlementDate(22, QuantLib::September, 2004);
settlementDate = calendar.adjust(settlementDate);

dpf = TheParticipantFactoryWithArgs(argc, argv);

qldds_utils::BasicDomainParticipant irs_participant( dpf, IRS_DOMAIN_ID );
irs_participant.createSubscriber();

QuantLibAddinCpp::qlLibor("Libor", "USD", "6M", "", false, false, true);

DDS::Topic_var swap_rate_helper2_topic =
                        irs_participant.createTopicAndRegisterType
                        < ratehelpers::qlSwapRateHelper2TypeSupport_var,
                       ratehelpers::qlSwapRateHelper2TypeSupportImpl >
                       (  SWAP_RATE_HELPER2_TOPIC_NAME );
```

```
// This lock synchronizes QuantLib calls
ACE_Mutex qldds_lock;

irs_participant.createDataReaderListener<ratehelpers::qlSwapRateHelper2
DataReaderListenerImpl>
        ( qldds_lock, swap_rate_helper2_topic );
```

Configuring and Building QLDDS

QLDDS can be configured using configure.py located in the root directory of the package. This script creates an environment file with all the necessary variables for this package.

Usage:

Unix (Assuming all dependences were installed under $HOME):

```
$ python configure.py --dds_dir=$HOME/DDS --boost_dir=$HOME
--quantlib_dir=$HOME --oh_dir=$HOME --qladdin_dir=$HOME
```

Windows, Visual Studio Command Prompt (Assuming all dependences were installed in their default locations):

```
C:\qldds>python configure.py --dds_dir=c:\DDS
--boost_dir=c:\progra~1\boost\boost_1_51 --quantlib_dir=c:\QuantLib-1.2.1
--oh_dir=c:\build_ql_1_2_0\ObjectHandler
--qladdin_dir=c:\build_ql_1_2_0\QuantLibAddin
```

Upon the successful execution of the configuration script the following file with environment variables will be created in the root directory of the package: Windows - qldds_env.bat, Unix - qldds_env.sh . This file should be sourced before building other components of the package.

Auto Generation

AutoGeneration of DataTypes and DataReaders

$QLDDS_HOM/gensrc/qldds_gensrc.py is a script that converts QuantLibAddin for C++ XML function definitions to IDL and creates default implementation of DataReaders for converted types. This script requires locations of **source** directories of ObjectHander, GenSrc and QuantLibAddin that can be passed as command arguments.

Usage:

Unix(assuming all dependences are located at $HOME):

```
$ python qldds_gensrc.py --oh_dir=$HOME/ObjectHandler-1.2.0
--gensrc_dir=$HOME/gensrc-1.2.0
--qladdin_dir=$HOME/QuantLibAddin-1.2.0
```

Windows:

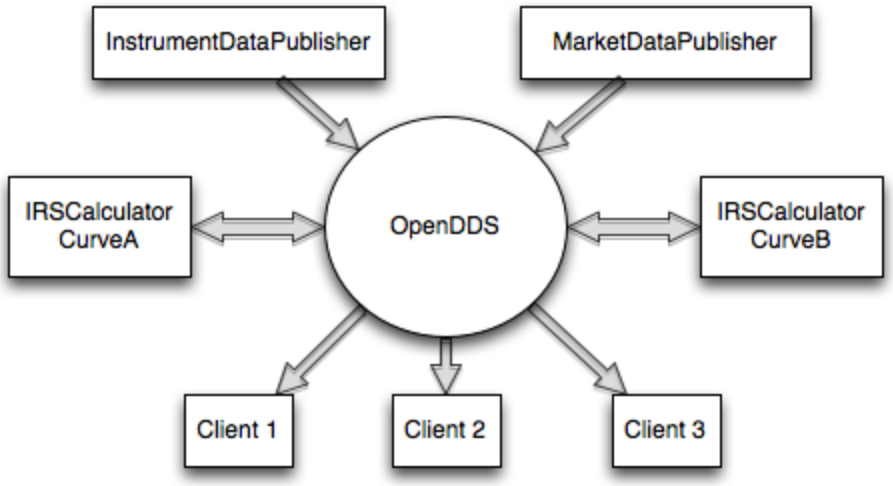
```
C:\qldds\gensrc>python qldds_gensrc.py
--oh_dir=C:\build_ql_1_2_0\ObjectHandler
--gensrc_dir=c:\build_ql_1_2_0\gensrc
--qladdin_dir=c:\build_ql_1_2_0\QuantLibAddin
```

Examples

Interest Rate Swaps

Scenario:

InstrumentDataPublisher publishes interest rate swap contract information. MarketDataPublisher publishes components of swap curves that include Deposits, FRAs and Swaps. Two calculators subscribe to swap contract information and market data, reprice received swaps against the latest market data and republish priced portfolio of swaps to clients. *(Fig 3.0)*



(Fig 3.0)

InstrumentPublishesher publishes contract information at a rate of 100 contracts per second. Swap contract information consists of *schedule::qlSchedule* and *vanillaswap::qlVanillaSwap* QLDDS data types. Upon publishing each of the samples of a

given data type, publisher waits for an acknowledgement from each of the subscribers/IRSCalculators.

Market Data Publisher continuously publishes updates on the instrument components of a swap curve that include samples of *ratehelpers::qlDepositRateHelper2, ratehelpers::qlFraRateHelper2 and ratehelpers::qlSwapRateHelper2* QLDDS data types.

IRSCalculators subscribe to all listed above data types. Once per second, they reprice received interest rate swaps against the specified in command line curve. Priced interest rate swaps are subsequently published in form of a portfolio to three clients.

Clients subscribe to priced swap portfolio data and display portfolio NPVs on the screen.

Topics:

Topic Name	Topic Type	Source	Destination
SCHEDULE_TOPIC	schedule:: qlSchedule	Instrument DataPublisher	IRSCalculator
VANILLA_SWAP_TOPIC	vanillaswap:: qlVanillaSwap	Instrument DataPublisher	IRSCalculator
DEPOSIT_RATE_HELPER2_ OPIC	ratehelpers:: qlDepositRateHelper2	MarketData Publisher	IRSCalculator
FRA_RATE_HELPER2_TOPI C	ratehelpers:: qlFraRateHelper2	MarketData Publisher	IRSCalculator
SWAP_RATE_HELPER2_TO PIC	ratehelpers:: qlSwapRateHelper2	MarketData Publisher	IRSCalculator
IRS_PORTFOLIO_TOPIC	IRS::Portfolio	Client	Client

Configuring and Compiling:

Unix:

```
$ $ACE_ROOT/bin/mwc.pl -type gnuace InterestRateSwaps.mwc
$ make
```

Windows(Visual Studio Command Prompt 2008):

```
> %ACE_ROOT%\bin\mwc.pl -type vc9 InterestRateSwaps_vc9.mwc
> msbuild.exe InterestRateSwaps_vc9.sln
```

Start scripts:

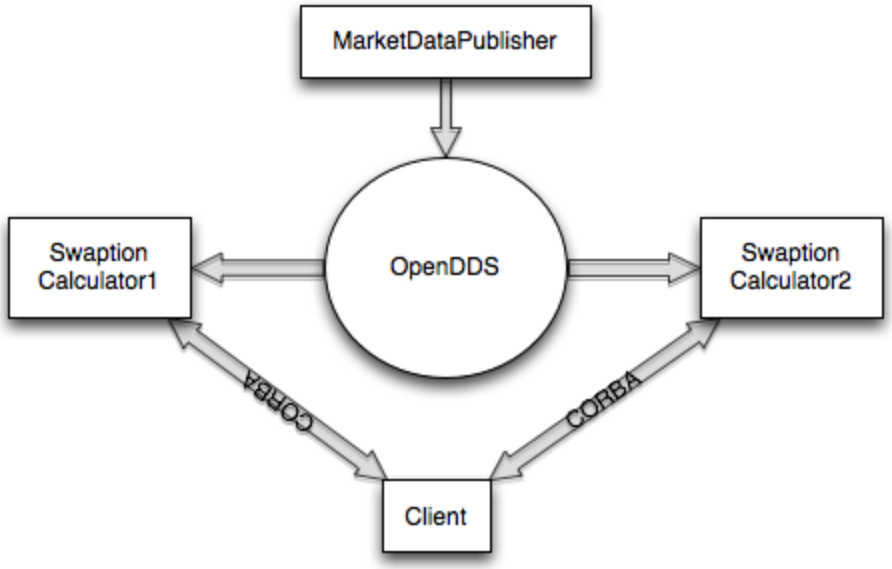
Unix : run_test.pl
Windows : run_test.bat

DDS and CORBA

Swaptions

Scenario:

MarketDataPublisher publishes updates on the volatility surface and market data components of a swap curve. Curve Market Data consists of Deposits, FRAs and SwapRates. Two Swaption calculators consume volatility and curve data. Client is continuously making synchronous calls to calculators to compute NPV on the provided Swaption indicative data. That data consists of tenors of an option and an underlying swap, as well as curve and surface names. This example demonstrates the usage of OpenDDS in conjunction with CORBA. *(See Fig 4.0)*



(Fig 4.0)

Topics:

Configuring and Compiling:
Unix:

```
$ $ACE_ROOT/bin/mwc.pl -type gnuace Swaptions.mwc
$ make
```

Windows(Visual Studio Command Prompt 2008):

```
> %ACE_ROOT%\bin\mwc.pl -type vc9 Swaptions_vc9.mwc
> msbuild.exe Swaptions_vc9.sln
```

Start scripts:

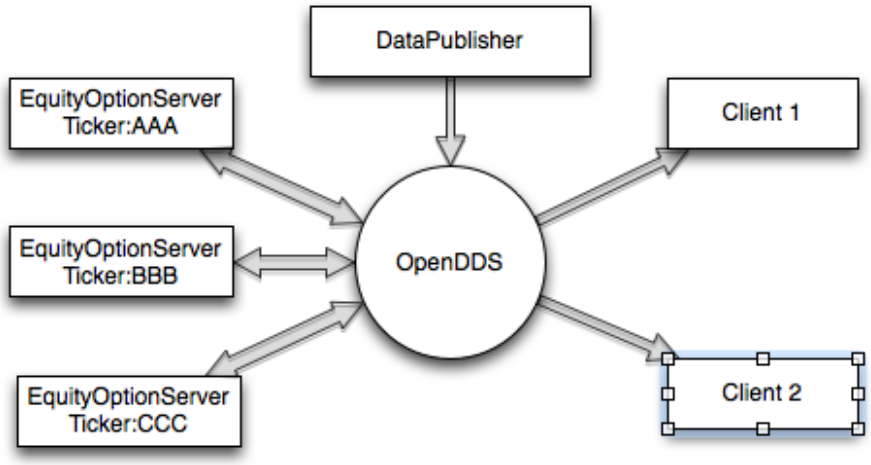
Unix : run_test.pl
Windows : run_test.bat

MultiTopic Subsciption

Equity Options

Scenario:

DataPublisher publishes indicative and market data information for 20 put and call strikes for three stock tickers. Three calculators(EquityOptionServers) subscribe to indicative and market data for a given stock ticker. They compute NPVs and Greeks on puts and calls of received strikes and publish results to clients. Two clients receive data published by 3 calculators and display the results. *(See Fig 5.0)* This example also demonstrates the multi-topic subscription feature of OpenDDS by having sequences of BlackContantVols, GeneralizedBlackScholesProcesses, StickedTypePayoffs and EuropeanExcercises published by the DataPublisher and joined on the ticker by the EquityOptionServer.



(Fig 5.0)

Topics:

Configuring and Compiling:
Unix:

```
$ $ACE_ROOT/bin/mwc.pl -type gnuace EquityOptions.mwc
$ make
```

Windows(Visual Studio Command Prompt 2008):

```
> %ACE_ROOT%\bin\mwc.pl -type vc9 EquityOptions_vc9.mwc
> msbuild.exe EquityOptions_vc9.sln
```

Start scripts:

Unix : run_test.pl
Windows : run_test.bat